Lund University
Computer Science
Jesper Öqvist, Niklas Fors, Görel Hedin

Programming Assignment 3
Visitors, Aspects, and Attribute Grammars

This assignment will give you experience in using two techniques for imperative computations in a compiler: *static Aspect Oriented Programming* (static AOP) and the *Visitor Pattern*. This assignment will also introduce *Attribute grammars*.

To understand visitors and static AOP, there is a new demonstration example, CalcComp. You will then extend your SimpliC implementation from assignment 2 with a visitor-based computation of *Maximal Statement Nesting*, and static AOP-based computations of *pretty printing* and *name analysis*. As an introduction to attribute grammars, you will work on a demonstration example called MinTree.

Nota bene! You can only use Java 8 constructs in your compiler code. This is because the JastAdd tool currently does not support newer versions of Java. This means that you cannot use, e.g., Local Variable Type Inference from Java 10[1], or Switch Expressions from Java 14[2]. While you can use Java 11 and 17 compilers and jvms, the gradle build script checks that the source code is compatible with Java 8.

Try to solve all parts of this assignment before going to the lab session. If you get stuck, use the forum to ask for help. If this does not help, you will have to ask at the lab session. Make notes about answers to questions in the tasks, and about things you would like to discuss at the lab session.

▶ Major tasks are marked with a black triangle, like this.

1 The CalcComp Demo

The CalcComp demo project shows how to implement a simple code metric using visitors, and simple error checking and pretty-printing using static AOP.

▶ Download the CalcComp example and run the tests, so you see it works.

1.1 Visitors

▶ Study the `Visitor` interface and `accept` methods in `src/jastadd/Visitor.jrag`.[3] Note that the Visitor interface has one `visit` method for each concrete AST class, and that each concrete AST class has an `accept` method. What does each `accept` method do?

▶ Study the class `TraversingVisitor` in `src/java/lang`. This class implements the `Visitor` interface. What does each `visit` method do?

▶ Study the class `CheckInteractiveVisitor.java` in `src/java/lang` which implements an example Visitor. The visitor checks if there are any interactive statements (`Ask`-statements) in a Calc program. Note that there is a static method called `result` that takes an AST parameter, and a visit method for the `Ask` class. Try to answer the following questions:

- The `main` method in `Compiler.java` calls the `result` method with the program AST as the argument. What happens when the `result` method is called? How is double dispatch used?
- Suppose you wanted to count the number of `Ask` statements instead. How would you change the visitor?

[1] Local Variable Type Inference: `https://openjdk.org/jeps/286`
[2] Switch Expressions: `https://openjdk.org/jeps/361`
[3] The code in Visitor.jrag is boilerplate code that would typically be generated automatically by some tool. However, for this assignment, we think it is a good idea to write it manually, so you get a better understanding of how visitors work.

1.2 Pretty printing using aspects

With JastAdd, it is possible to use static aspects instead of visitors. Aspects allow methods to be added directly to existing AST classes, and there is no need for the double dispatch that visitors use. We will now look at how *pretty printing* (also known as *unparsing*) is implemented in CalcComp using aspects.

Pretty printing is the programmatic printing of an AST back to its corresponding textual form. This can be useful in several situations, for example when printing error messages, or when debugging a compiler.

▶ Study the aspect `PrettyPrint.jrag` in `src/jastadd`. Some things to note:

- There is a method `prettyPrint(PrintStream)` that is intended to be called by a client, e.g., the main program.
- `System.out` has the type `PrintStream`, so the client can pass `System.out` as the argument to `prettyPrint`.
- There are methods `prettyPrint(PrintStream, String)` that are used internally to do the pretty printing. The `String` argument is used for adding a suitable number of blanks as indentation.

1.3 Name analysis and error checking

The name analysis in CalcComp checks that variables are declared before they are used. For example, the following code is not valid because the name `b` is used before it has been declared.

```
let
    a = b
    b = 3.0
in
    a
end
```

Multiple declarations of the same name in the same `let` statement are not allowed. However, "shadowing" is supported, so that a declaration in a `let` statement shadows outer declarations of the same name. For example, the following is valid because the second declaration of `b` is inside another `let` statement:

```
let
    a = 1.0
    b = a
in
    let
        b = 2.0
    in
        b
    end
end
```

The name analysis in CalcComp is implemented using a stack of symbol tables. The tree is traversed, and each time a new scope is entered, such as a `let` statement, a new `SymbolTable` instance is pushed on top of the stack.

▶ Study the `NameAnalysis.jrag` aspect in `src/jastadd`. Things to note, and questions to answer:

- The aspect contains a class `SymbolTable` with methods `declare`, `lookup`, and `push`.
- The aspect contains a method `Program.checkNames(PrintStream)` that checks a program for name analysis errors and prints any error messages on the PrintStream.

- To implement `checkNames(PrintStream)`, each AST class has a method `checkNames(PrintStream, SymbolTable)`. These methods print any error messages to the `PrintStream`, add any declarations to the current `SymbolTable`, push nested `SymbolTables` as needed, and traverse recursively into any subtrees. Make sure you understand how these methods work.
- The symbol table is simpler than a traditional symbol table in that it only keeps track of names and not the binding (declaration) of a name. This would be needed to support, for example, type checking. How would you modify the `SymbolTable` class to support name binding?
- Suppose you have a main program with a `Program` AST. What would a call look like that performs name analysis and prints the errors to standard error? (Hint: google *standard streams java* if you don't know what standard error is.)
- How does the push operation work in `SymbolTable`? Why is there no pop operation?

.

2 SimpliC

You will now extend the compiler for SimpliC that you implemented in Assignment 2 with some analyses using visitors and aspects.

▶ Make sure that your code from Assignment 2 is in a consistent state that builds and tests correctly. Copy the files to the `A3` subdirectory, and continue working there.

2.1 Maximal Statement Nesting for SimpliC

Maximal Statement Nesting (MSN) is a simple metric which computes the maximum nesting depth of statements in a program. If there are no nested statements then MSN=1, if there is one statement nested inside another then MSN=2 etc. The MSN of the example below is 3:

```
int main() {
  // depth = 1
  int i = 100;
  while (i > 0) {
    // depth = 2
    if (i == 5) {
      // depth = 3
      print(i);
    }
    i = i - 1;
  }
  if (i == 0) {
    // depth = 2
    print(i);
  }
  // depth = 1
  return 0;
}
```

▶ Implement the visitor framework for SimpliC, including a `TraversingVisitor` class.[4]

▶ Think about how you could implement MSN analysis for SimpliC. Think first how you would solve it if you added methods to the AST classes. Then think about how you would solve it using a visitor. Try to answer the following questions.

- What are possible strategies for implementing the MSN analysis? Should you use state variables inside the visitor or the data parameter? What are the pros and cons of these approaches?
- Suppose the visitor framework had type parameters for the return value and the data parameter. What would the advantage be?

▶ Implement a visitor to compute the MSN of a SimpliC program, including automated tests with examples where MSN is 1, 2, and 3, respectively.

2.2 Pretty printing for SimpliC

▶ Implement pretty printing of SimpliC programs using static aspects. Your code should be structured similarly to the pretty printing in the CalcComp demo (see section 1.2), and include automated tests. For syntactically correct programs, the test input should be the same as the expected output. However, in case you do not store parenthesis expressions in your AST, you may ignore pretty printing the parentheses, for simplicity.[5]

2.3 Name analysis and error checking for SimpliC

You should implement name analysis and error checking of two basic types of errors:

- Check and report errors for multiply declared functions and variables
- Check and report errors for uses of undeclared functions and variables

The declaration order is important. Functions and variables must be declared before they are used. For example:

```
int a() {
  return b(); // error: b is not yet declared
}

int b() {
  return 1;
}
```

Declare-before-use for variables:

```
int main() {
  int a;
  a = 2 + b; // error: b is not yet declared
  int b = 3;
  return a;
}
```

[4]Note that if you use a tool that supports visitors, it would typically generate the framework code for you. You will write this code yourself in order to better understand how it works, even if it is a bit repetitive.

[5]Pretty-printing a minimal number of parentheses would be a bit too much work for this assignment. If you want to do it, it is better to wait until you have learnt how to use attributes (Assignment 4).

The variable names are allowed to be shadowed, if declared inside another statement:

```
int main() {
  int a = 3;
  if (a != 0) {
    int a = 4; // okay: not in same block
    return a;
  }
  int a = 5; // error: redeclaration
  return a;
}
```

Function parameters should of course also be considered in the name analysis:

```
int main(int a) {
  return a; // okay: a refers to the parameter
}
```

The name errors are checked by implementing a simple form of name analysis. You will not need to bind variable uses to their declarations so you can use a very simple symbol table as in the CalcComp demo.

When implementing the checks for variable and function names, it is useful to have `IdDecl` and `IdUse` AST nodes for *both* variable and function declarations/uses. The name analysis can then handle all named entities in the same way, and does not need to distinguish between variables and functions. To check that a variable is not used in a position where a function is expected, or the other way around, will be easy to add later on as part of the type checking. `IdDecl` and `IdUse` are used in the CalcComp demo, although in the CalcComp case there are no functions.

▶ Think through how you are going to implement name analysis and error checking. Try to answer the following questions.

- How can you handle built-in functions like `print` and `read`? (*Hint!* You can initialize the symbol table with mock declarations of built-in functions.)
- In what circumstances should you be able to re-declare, or shadow, a variable name?
- In your opinion, should it be possible to shadow function parameters? Why, or why not?
- The examples provided above are good test cases. What other important test cases can you think of? Consider multiple parameters, if-then-else, etc.

▶ Implement name analysis and the basic error checking described above, including good automated tests.

A note on line and column numbers

The scanner can set the line and column number of each scanned symbol, by using the metavariables `yyline` and `yycolumn`. See `scanner.jflex` in one of the example projects. During parsing, the line and column information is carried over to the AST nodes, and can be accessed for each AST node by the calls `getLine(getStart())` and `getColumn(getStart())`. See `src/jastadd/NameAnalysis.jrag` in the CalcComp demo for an example. However, Beaver only sets these values for the returned production of a rule. So if one of your rules creates more than one node, the line and column numbers will work correctly only for the returned node. Therefore, if you get the value `0` for rows and columns, try refactoring your Beaver rules so that each new node is returned by a separate beaver rule.

Setting correct line and column numbers is important for being able to give suitable error messages. Furthermore, in the labs, you will be using a debug tool called CodeProber (section 4.1), for which correct line and column numbers are also important.

A note on `System.exit(1)`

Usually, a program will exit with a non-zero error code in case of errors. This can be done by calling `System.exit(1)` (to exit with error code `1`). The self grader uses the exit code to determine if the compilation ends with an error or not. To use the self grader, you will need to adapt the main program to exit with non-zero when there are semantic errors like name binding errors.

3 MinTree - Introduction to Attribute Grammars

You will now get an introduction to Attribute Grammars, using an example called MinTree. Next week's assignment will go deeper into attribute grammars, and apply them for semantic analysis of SimpliC.

MinTree is a small language that describes trees of numbers, and your task is to compute the minimum number in the tree, using attribute grammars. The MinTree project is provided on the course web page. The project contains an abstract grammar, pretty printing, a main class, and test cases.

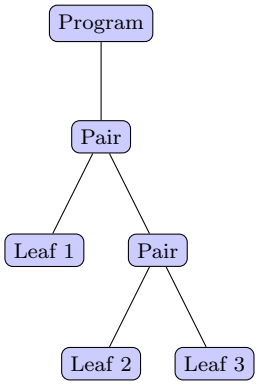
The abstract grammar of MinTree is defined as follows.

```
Program ::= Node;
abstract Node;
Pair : Node ::= Left:Node Right:Node;
Leaf : Node ::= <Number:int>;
```

The class `Program` represents the root node of the AST. The abstract class `Node` with two concrete subclasses, `Pair` and `Leaf`, models the recursive tree structure. Using this grammar, we can create an AST explicitly, without any parser, as follows.

```
new Program(new Pair(new Leaf(1), new Pair(new Leaf(2), new Leaf(3))))
```

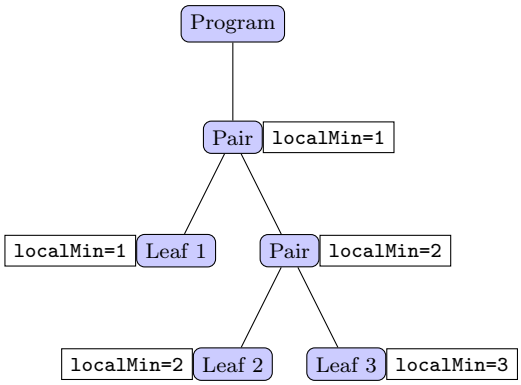
This expression can be visualized as follows.



- Download the MinTree project. Build the project and run the test cases (`./gradlew test`). All test cases should fail at this point. Also, build and run the main `lang.Compiler` program that prints out the tree above in a textual format. To build it as a Jar file, use `./gradlew jar`. To run the Jar file, type `java -jar compiler.jar`.
- Rename the project to `A3-MinTree-solution`. Place it as a subdirectory of your repository, and put it under version control (`git add A3-MinTree-solution/`).

3.1 The localMin attribute

Attributes are *computed properties* of AST nodes that are defined by *equations*. How can we define the minimum value of the AST using attributes? Recall that `Node` is the superclass of `Leaf` and `Pair`. Let's introduce an attribute `localMin` for `Node` that is the minimum value of the `Node`'s subtree. I.e., we would like the following attributed tree:



We can see that the value of `localMin` for `Pair` nodes is the minimum value of its children's `localMin`.

We can declare `localMin` as an attribute on the abstract class `Node`, so both subclasses `Pair` and `Leaf` get the attribute:

```
syn int Node.localMin();
```

This is a **syn**thesized attribute with the type `int` on class `Node` and with the name `localMin`. The attribute has been declared as a synthesized attribute, which means the equation must be located in the same `Node` object (not in an ancestor). We can provide one equation in class `Pair` and another in class `Leaf` to take care of the two kinds of `Node`s. The equations can use tokens and attributes of the node and its children.

An equation for the class `Leaf` can be specified as follows.

```
eq Leaf.localMin() = ...;
```

Here, the right-hand side of the equation is an expression. This kind of equation is called *expression-style equation*. For more complicated equations, the equation can be specified as a block. The above equation is equivalent to the following *block-style equation*.

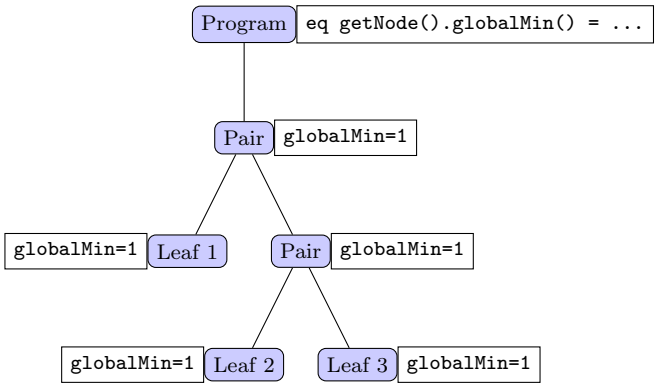
```
eq Leaf.localMin() {
  ...
  return ...;
}
```

▶ Define the attribute `localMin` for the classes `Leaf` and `Pair` by adding two equations in the aspect file `MinValue.jrag`. Use the expression-style equation for the class `Leaf` and block-style equation for the class `Pair`. Verify that the test case for `localMin` passes.

Note! If you are using VS Code as your editor, there is a syntax highlighting extension for JastAdd that you can install. See 4.2.

3.2 The globalMin attribute

We have now synthesized the information upwards in the tree. We will now let all nodes know what number is the mininum number. We can do this by using an *inherited* attribute, that is, an attribute whose equation is defined by an ancestor node in the AST. This is illustrated in the following figure.



Here, the class `Node`, and thus `Pair` and `Leaf`, has an inherited attribute `globalMin`, but the *equation* is defined by the `Program` node, that is, an ancestor in the AST. An **inh**erited attribute can be declared as follows.

```
inh int Node.globalMin();
```

And the equation is specified as follows.

```
eq Program.getNode().globalMin() = ...;
```

Note that we specify the child (`getNode`) for which the equation is valid, according to the names in the abstract grammar. The equation above states that the value of the attribute `globalMin` is `...` for the child `Node`. The code `...` executes in the context of `Program`, and can access attributes, tokens, and children in `Program`.

The equation actually applies not only to `getNode`, but to *all* nodes *in the whole subtree* of `getNode` that happen to have a `globalMin` attribute. This is a mechanism called *broadcasting*, that is very useful in order to avoid having to define a lot of so called *copy* attributes that simply copy a value from a parent to a child.

If we want the equation to be valid not just for the subtree of a particular child, but for *all* children, we can specify `getChild` instead of `getNode`, which in this case does not matter since `Program` has only one child.

- The `MinValue.jrag` aspect contains an equation for `globalMin` that simply defines it to be 0. Change the equation so that the correct value is computed. Verify that the corresponding test case passes.
- There is an attribute `isMinValue` on `Leaf` that is supposed to tell if the node is the minimum number. Change the equation for the attribute to compute the correct value. Verify that the test case for `isMinValue` passes.
- The numbers in the tree do not need to be unique. This means that several `Leaf` nodes can have the same number. We will now compute how many `Leaf` nodes that have the minimum number. Define the attribute `nbrOfMinValues` for both the class `Program` (computing the global value) and for `Node` (computing the local value for that subtree). Verify that all test cases pass. Also, add a new test case where the value of `nbrOfMinValues` is larger than 1.

3.3 Side-effect free equations

For attribute grammars to work, the equations must be free from externally observable side effects, i.e., effects that other equations can see. This means that computing the equation several times should yield the same result. For example, the equation code must never change global variables that other equations might use (or call methods that do so). Note, however, that it is fine for an equation block to introduce local variables and assign to them, because these variables are not accessible from outside the block—the side effects on those variables are not externally observable.

Can I add print statements to the equation code? Yes, you can do that, but only for debugging, e.g., to see if an attribute is evaluated or not. You cannot use it for production code. The reason is that you do not control the order in which equations are executed—this is decided by an attribute evaluation engine. If you want to print things as part of the compilation, say, to prettyprint an AST, you will instead use normal methods, and which may access attributes in order to easily print computed information, such as type-checking errors.

3.4 Using attributes from methods

▶ There is an aspect `PrettyPrint.jadd` with methods for prettyprinting a MinTree, and that is called from the `lang.Compiler` class. Change the prettyprinter to print `*** MINIMUM ***` for each leaf node that has the minimum value. Check that it works by running the main `lang.Compiler` program again.

4 Development tooling for JastAdd

You will now try out a very useful development tool for JastAdd: CodeProber. Through CodeProber you can inspect the attribute values of your compiler. We also mention a syntax highlighter that is useful if you are using the VS code editor.

4.1 CodeProber—probing AST node properties from a code editor

CodeProber is a web-based code editor that allows you to interactively explore attributes and other properties of the code. With CodeProber, you point in the code to identify an AST node, and you can then create a so called *probe* that shows the value of a property. If you edit the code, the probe will update, always showing the current value.

CodeProber uses the compiler to parse the edited code, so it is less useful for MinTree which lacks a parser. However, CodeProber will be very convenient for using on your SimpliC compiler.

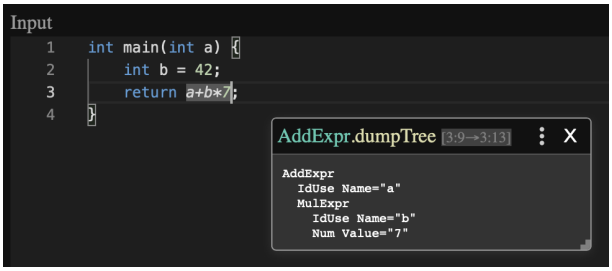
CodeProber is being developed since 2022 in a research project at the department, and suggestions for enhancements are welcome.

▶ Try out CodeProber on your SimpliC compiler using the following steps:

- Make sure your compiler is prepared for running with CodeProber, by having a static method `CodeProber_parse` defined in the main program, and that returns the parsed AST. See the main program `Compiler.java` of the CalcComp demo project for an example.
- Download `code-prober.jar` from the following repository:
  `https://github.com/lu-cs-sde/codeprober/`
  (Click on the `code-prober.jar` file, and press the download button to the far right in the window).
- Place the `code-prober.jar` file in a suitable place, e.g., in your main directory for the course. (Don't add it to the git repo, however.)

- The tool works by running a local web server and opening a web page on localhost. To start the server, run code-prober with your compiler as the argument. If you are in, say, the A2 directory, and `code-prober.jar` is one level up, you can write:
  ```
  java -jar ../code-prober.jar compiler.jar
  ```
  where `compiler.jar` is the compiler you built for SimpliC.
- To start the code editor, open a browser on the page `http://localhost:8000`.
- If you run CodeProber on Windows via WSL, you may need to open the browser using the URL written by CodeProber on the console, which includes an authorization token. The URL will look something like: `http://localhost:8000?auth=key-MS4...`
- Write some SimpliC code in the code editor, e.g., copy one of your test programs.
- Create a probe:
  - Right-click to select `Create Probe`.
  - Select one of the nodes in the menu (all AST nodes covering the position you clicked are shown in the menu). Note that when you hover over the nodes in the menu, the corresponding code lights up in the editor.
  - A menu with methods of the selected node appears. Select the `dumpTree()` method.

  Note that the probe shows the value of `dumpTree()` for the selected node, as in the example below. Now edit the program and watch how the value of the probed method changes. In future assignments, you will be able to inspect attribute values in a similar way.

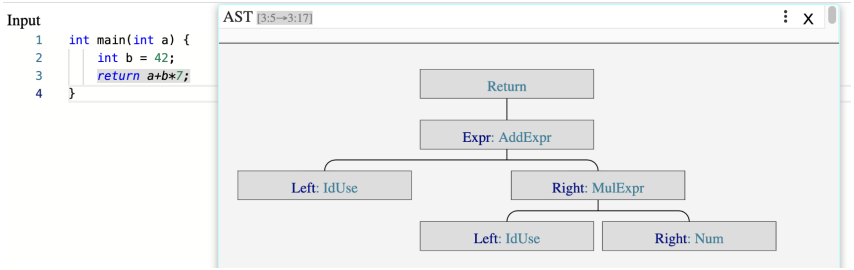


▶ You might want to change a couple of settings in the menu in the right-hand side pane:

- Click/unclick `Dark mode` to get dark or light mode.
- Select `C` in the `Syntax highlighting` menu, to get highlighting more suited for SimpliC.
- You can ignore the other settings.

▶ CodeProber can also show a visual representation of the AST.

- Start creating a probe for one of the nodes in the program, for example a return statement.
- Instead of selecting one of the methods, select the three vertical dots to the right and select `Render AST downwards`.
- Try hovering over the nodes in the tree to see what parts of the program they correspond to.
- Try editing the code to see how the tree changes.



The methods that can be probed are those without arguments or with arguments of primitive or ASTNode types. Later in the course, when you use attributes to define static semantics and code generation, you will be able to probe them in this way.

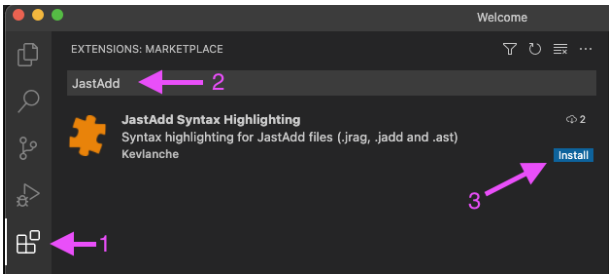
If you want to, you *can* actually run CodeProber on MinTree and look at its AST and its attributes. However, since MinTree doesn't parse anything, the edited code is simply ignored. You can click in the code window to create a probe on the root, then create an AST view, and then click on the different nodes to create probes to explore their attributes.

For more information on CodeProber, click on its `Help` button, and see also `https://github.com/lu-cs-sde/codeprober/`.

4.2 JastAdd syntax highlighting in VS Code

If you are using VS Code as your editor, you can install syntax highlighting for JastAdd files (.ast, .jrag, and .jadd) as follows:

1. Open VS Code and click the extensions tab
2. Search for "JastAdd"
3. Click "Install"



Alternatively, you can install it via your browser here: `https://marketplace.visualstudio.com/items?itemName=Kevlanche.jastadd-syntax-highlighting`

5 What to show and discuss with your supervisor

When you are ready with the assignment, these are typical things your supervisor may ask you to do:

- Show your MSN visitor for SimpliC, and the automated tests you used for testing it.
- Show your pretty-print aspect for SimpliC, and your automated tests for it.
- Show your name analysis aspect for SimpliC, and your automated tests for it.
- Show your attribute grammar computing `localMin`, `globalMin`, and `nbrOfMinValues`. Show that the tests pass. Show that you can run the MinTree compiler so that it prints the text
  ```
  ***MINIMUM ***
  ```
  in the right way.
- What was your experience with CodeProber?