AST

==========

01-04-24:

Creating a Control-Flow-Graph(CFG) from an Abstract Syntax Tree (AST)

- we will not be creating a CFG and instead we will use the AST

In TACO, `BlockSimplifier.java` has the structure of the Visitor Pattern we need.

Idea:

There is an invariant in the way the visitor works, especially there is a stack which will never have one thing. Whenever there is a statement with a condition and body, take the body (self (while loop)), pass as parameter(this) in `accept`. The body will be visited by the same visitor. Afterwards, we care about the condition. The `BlockSimplifier` doesn't take care of the conditions in the loop. In our case, we call the visitor on the condition as well.

Generate a new while statement and push it onto the stack. Input while statement and output modified while statement. Two recursive calls on if statement, with branches, and push new if statement onto stack. Same for a for loop.

`JmlAstClonerStatementVisitor` is the most general visitor method. Check `visitForStatement` method in `BlockSimplifier.java`, although it doesn't modify the condition. Also `ASTSimplifierManager.java` is the simplify method. First simplifier that will be executed is `BlockSimplifier.java`. Put a breakpoint in the for loop and look at what `BlockSimplifier.java` does. **Look for Visitor visiting AST**. Maybe modify visitor.

Only modify while, if, and for statements. Modifies things in one problem. What we want is two make at most two problems to analyze. Enter while loop, copy while loop and enter the body. Another one where we skip the while loop. Modify visitor methods and rather than returning the stack, we return a queue of stacks. Initially call with empty queue. In the while statement, generate two programs. Queue is a parameter, and the while statement as another. How do we make use of this?

01-05-24:

From yesterdays meeting:

> Visitor Pattern: Traversing something, it is done recursively. We know there is an if statement, u. If there is an if statement, there is a condition, if branch and else branch. There should be a soot method that gives access to branching condition for if and else branch. Keep track of the surrounding code, for example if there is no if or else condition.

From Marcelo on Teams:

> Make a new TACO project in which we can add your code. Create in the new project a class `JmlAstDeterminizerVisitor` by copying class `JmlAstClonerStatementVisitor`. The idea is

> that whenever this visitor is invoked on a method will remove one source of branching and produce new programs. The new **Visitor** requires a new attribute (field) named "*programs*" that has type `ArrayQueue<Stack<Object>>()`. Now, the idea is that whenever you call `visitIfStatement` with an if Statement *if (cond) then B1 else B2* you have to generate two programs. **First Program**: "`assert cond; B1`" **Second Program**: "`assert !cond; B2`" Since we are going to remove only 1 branching, up to this point we know that the Queue has length one. We then store the first Program in the top of the first stack, and afterwards create a new stack, store the Second Program, and queue the new stack into the queue. And this solves the if case. Let us see the while case. It is exactly the same, with the differences: The input is a `WhileStatement` of the form "`while (cond) {B}`" We then create 2 programs: **First Program**: "`assert cond; B; while (cond) {B}`" **Second Program**: "`assert !cond;`" We will leave for loops for the future (they are easy to handle by transforming for lops into while loops and reducing nondeterminism from the resulting while loop.) For all other methods, the intuition is that the queue may have length 2, and therefore we need to clone the particular statement in each version of the program. Una subtlety.... Before splitting the programs in the treatment of *if* and *while*, **we need to make sure the queue has size 1**. Otherwise, we will end up reducing nondeterminism more than once. Let us go back to statements other than *if*, *while*. Let us consider as an example, method: `visitCompoundStatement` This method handles sequential compositions of the form `St1;St2;...;Stn`. The method receives as input an array with statements [St1,St2,...,Stn]. If `Sti` is an if statement, which will be split into two different pieces of code in the Queue (namely FP (for First Program) and SP, ideally we want to return in the queue: `[[St1,St2,...FP,St_{i+1},...,Stn], [St1,St2,...SP,St_{i+1},...,Stn]]` One way to do this is in the while loop in the code for `visitCompoundStatement`, we can generate two arrays with cloned versions of `St1,...St_{i-i}` and if an if or while appears, we copy FP in one array and SP in the other, and continue with the remaining statements duplicating them in both arrays.

Summary of Marcelo's Note

- Stacks of Queues
- Stacks contain branches/programs
- Queue holds the Stacks of branches/programs
- [**NOTE**] **Make sure *Queue* size is length *1*** so that nondeterminism isn't reduced more than once

1. First, make a copy of TACO
2. Create a new class: `JmAstDeterminizerVisitor` based on the `JmlAstClonerStatementVisitor` class
    - This class will remove a source of branching and produce new programs
3. Add attribute *programs* with the type `ArrayQueue<Stack<Object>>()`

If Statement

1. For the **if statement**, when `visitIfStatement` is called in the form of `if(cond then B1 else B2`:

```
if(cond){
        B1 // then Branch 1
}
```

```
else{
        B2 //else Branch 2
}
```

2. `visitIfStatement` should generate two programs:
    1. `assert cond; B1` -> if statement branch
    2. `assert !cond; B2` -> else statement branch
3. In this case, there will be already a length of one for the queue since there is at least one branch

While Statement

1. For the **while statement**, when `visitIfStatement` is called in the form of `while(cond) {B}`:

```
while(cond){
    B // Branch
}
```

2. We then make two programs:
    1. `assert cond; B; while (cond) {B}`
    2. `assert !cond;`

Other Statements

1. Create method `visitCompoundStatement`, which handles sequential compositions of the form:
    - `St1;St2;...;Stn`
2. This method receives an input array with the statements:
    - `[St,St2,...,Stn]`
3. If St_i is an if statement, it will be split into two programs.
4. In the *Queue*, the if statement will be split into programs `FP` and `SP` (First Program and Second Program respectively), which will return: - `[[St1,St2,...FP,St_{i+1},...,Stn], [St1,St2,...SP,St_{i+1},...,Stn]]` 5. To do this, we generate two arrays with cloned versions of `[St1,...St_{i-i}]` and if an *if* or *while* appears, we copy **FP** in one array and **SP** in the other, and continue with the remaining statements duplicating them in both arrays, in the while loop.

[**NOTE**] *FP** is the first branch and **SP** is the second branch for the *if statement*

01-06-24

I will be using Mac from now to spend less time moving between Mac and WSL.

01-08-24

Creating TACO project copy to begin creating the visitor method.

New method located under utils directory, along with `JmlAstClonerStatementVisitor` and other related classes.

Working on understanding `JmlAstClonerStatementVisitor`, as well as what a *visitor pattern* is, how it works, and how I am going to implement it.

01-23-24

For the past couple of weeks, I have been researching how a Visitor Pattern works, as well as working out an example. My notes include everything I have learned by following a video as well as through other resources. Now, what I have to do is look at how this will be implemented in TACO under my copy of the project, and review the notes from before in order to traverse the AST.

01-25-24

Insert new class into `aJavaCodeSimplifier` -> compilation_units

02-01-24

Implemented `JmlAstDeterminzerVisitor` and ran TACO test successfully. Meeting today.

02-19-24

Looking into `JmlMethodDeclaration` usages, it looks like what we need is 2 bodies, one for the first program and another for the second.