

Jml Winter School 2008 Exercise 1

Adding initial support for while-loop invariant

prepared by Perry R. James and Patrice Chalin

Objective: add support for loop invariants to while loops, including lexing, parsing and RAC.

Setup: In addition to the instructions on the web site, you'll need to get specific versions of five of the projects:

```
org.eclipse.jdt.core 573,
org.eclipse.jdt.core.tests.{build,compile,model,performance} v_837.
```

1. Unit tests

- a. Before starting, let's write a unit test that fails.

2. Grammar file

- a. Open java.g. (open resource shortcut: Alt-Shift-R) If you have installed the gedit plugin, it should provide syntax highlighting.
- b. Find the rules defining a while statement by searching for 'while' (with the single quotes) (Find/replace shortcut: Ctrl-F). Notice that there are two grammar rules for while loops.
- c. Determine the changes to the productions needed. For this simplified introduction, pretend the relevant lines from the JML Reference Manual were

```
annotated-loop ::= [ loop-invariant ] while ( expression ) statement
loop-invariant ::= maintaining-keyword predicate ;
maintaining-keyword ::= loop_invariant
```

- d. Paste these in as comments and translate to JikesPG format. Note the use of camel case and that the semantic action is a single method call. After making this addition, there should be 4 rules for the while statement: the 2 original and 2 new ones.
- e. Add translations for the definitions of the remaining 2 non-terminals.
 - i. Note that there will be no semantic action for the definition of MaintainingKeyword, so an arrow (->) should be used instead of ::=.
 - ii. As part of the mechanism for not treating JML keywords as such in the Predicate, we need to add the non-terminal ExitJmlCause.
- f. After making the needed changes to the while loop rules, the grammar file should look like

```
WhileStatement ::= 'while' '(' Expression ')' Statement
/.$putCase consumeStatementWhile() ; $break ./
/.$readableName WhileStatement:/

-- <jml-start id="jml.loop-invariant" />
--annotated-loop ::= [ loop-invariant ] while ( expression ) statement
WhileStatement ::= LoopInvariant 'while' '(' Expression ')' Statement
/.$putCase consumeStatementWhileWithInvariant(); $break ./
/.$readableName WhileStatement:/
-- <jml-end id="jml.loop-invariant" />

WhileStatementNoShortIf ::= 'while' '(' Expression ')' StatementNoShortIf
/.$putCase consumeStatementWhile() ; $break ./
/.$readableName WhileStatement:/

-- <jml-start id="jml.loop-invariant" />
--annotated-loop ::= [ loop-invariant ] while ( expression ) statement
WhileStatementNoShortIf ::= LoopInvariant 'while' '(' Expression ')' StatementNoShortIf
/.$putCase consumeStatementWhileWithInvariant(); $break ./
/.$readableName WhileStatement:/
```

```

--loop-invariant ::= maintain-ing-keyword predicate ;
LoopInvariant ::= Maintain-ingKeyword Predicate ExitJml Clause ;'
/.$putCase consumeLoopInvariant() ; $break ./
/.$readableName LoopInvariant:/
--maintain-ing-keyword ::= loop-invariant
Maintain-ingKeyword -> 'loop-invariant'
/.$readableName Maintain-ingKeyword:/
--<jml -end id="jml.loop-invariant" />

```

- g. Fix the warning that "element 'loop_invariant' is not defined, assuming it is a terminal" by adding it to the \$Terminal s section, around line 110.
- h. Save the grammar file and regenerate the parser to confirm that there are no problems. The resulting parser should be LALR(1) and should have no Shift-Reduce or Reduce-Reduce conflicts. 3 errors in the parser are expected, since we haven't yet added the methods for the semantic actions.

3. Scanner

- a. Open Scanner.java (Open Type shortcut: Ctrl -Shift T) Select the one in org.eclipse.jdt.internal.compiler.parser.
- b. Add an entry for the loop_invariant keyword to the map from lexemes to terminal tokens, around line 3682. Note that generating the parser defined the needed TokenName loop_invariant.
- c. Save the file. That's all there is to adding a new keyword to the scanner. More work is needed in the parser, specifically in its consumeToken() method.

4. Parser

- a. Open Parser.java.
- b. Before adding the semantic actions, we should finish up with the new keyword, which is similar to that of the existing invariant keyword.
 - i. Search for TokenName invariant. It is only mentioned in consumeToken(), around line 9570.
 - ii. Add a case for TokenName loop_invariant. This will have the effect of putting the Scanner into a state in which JML keywords are treated as tokens as well as pushing the text of the token on the identifier stack.
- c. Now we're ready to implement the semantic action methods.
 - i. Use Quick Fix (shortcut: Ctrl -1) to add the declarations immediately after the consumeRule() method.
 - ii. Add the JML comments around the newly added code.

```

//<jml -start id="jml.loop-invariant" />
private void consumeStatementWhileWithInvariant() {
    // TODO Auto-generated method stub
}
private void consumeLoopInvariant() {
    // TODO Auto-generated method stub
}
//<jml -end id="jml.loop-invariant" />

```

- iii. Add the method body for consumeLoopInvariant().
 1. This will be very similar to that of consumeInvariantDeclaration().
 2. Copy-and-paste the body of consumeInvariantDeclaration()
 3. Update the comments for the grammar rule, the pre-identifierStack, and the post-astStack.

4. Change the type of `inv` to `JmlLoopInvariant`.
- iv. Define the class `JmlLoopInvariant`.
 1. Use Quick Fix to bring up the New Class Wizard.
 - a. Package: `org.jmlspecs.jml4.ast`
 - b. Superclass: `JmlClause`
 - c. Select "Constructors from superclass"
 - d. Finish
 2. Remove the TODO from the 3-argument constructor.
 3. Remove the 2-argument constructor.
 4. Navigate (F3) to the superclass and look at the `resolve()` and `analyzeCode()` methods.
 - a. `resolve()` resolves the type of predicate (and complains if it is not a boolean) as well as checks the conversions (in this case for auto-[un]boxing).
 - b. `analyzeCode()` does flow analysis checks on the predicate.
 5. To implement the `generateCheck()` method, we notice that it would be similar to that of `JmlInvariantForType`.
 - a. Copy-and-paste the method from `JmlInvariantForType`.
 - b. Question: It might have been easier to make this class a subclass of `JmlInvariantForType`. What are some reasons for and against this?
 - c. Navigate through `generateEvaluateAndThrowIfFalse()` until you come to the 3-argument version. Notice the use of the `CodeStream` to generate bytecode.
- v. Add the method body for `consumeStatementWhileInvariant()`.
 1. Go back to its declaration in the parser.
 2. Move this declaration to be after to the declaration of `consumeStatementWhile()`, keeping the `jml-start` and `-end` comments correct.
 3. Copy-and-paste the body of `consumeStatementWhile()` into `consumeStatementWhileInvariant()`.
 4. Update the comments to include the loop invariant.
 5. Notice that the original code places the `WhileStatement` in the same spot on the AST stack as the loop body.
 - a. Decrement the `astLengthPtr` before reading `Statement`.
 - b. Decrement the `astPtr` after reading `Statement` (just add the "--").
 6. Read the loop invariant of the AST stack, this time NOT decrementing its or the AST-length stacks' pointers.
 7. Change the constructor call from `WhileStatement` to `JmlWhileStatement`, and add the loop invariant as the first parameter.

```

//<jml-start id="jml.loop-invariant" />
private void consumeStatementWhileWithInvariant() {
    // WhileStatement ::= LoopInvariant 'while' '(' Expression ')' Statement
    // WhileStatementNoShortIf ::= LoopInvariant 'while' '(' Expression ')' StatementNo...

    this.expressionLengthPtr--;
    this.astLengthPtr--;
    Statement statement = (Statement) this.astStack[this.astPtr--];
    JmlLoopInvariant inv = (JmlLoopInvariant) this.astStack[this.astPtr];
    this.astStack[this.astPtr] =
        new JmlWhileStatement(
            inv,
            this.expressionStack[this.expressionPtr--],
            statement,
            this.intStack[this.intPtr--],
            this.endStatementPosition);
}
//<jml-end id="jml.loop-invariant" />

```

vi. Define the class `JmlWhileStatement`.

1. Make it a subclass of `WhileStatement`, as in 4.c.iv above.
2. Add a `JmlLoopInvariant` as the first formal parameter to the constructor.
3. Add "`this.invariant = invariant`" after the call to the super class's constructor.
4. Use Quick Fix to add the field, and make it final.
5. Use the Source -> Override/Implement methods feature (shortcut Alt-Shift-S) to add resolve and analyse methods from `WhileStatement`.
6. Implement the resolve and analyse methods.
 - a. Note that it is only necessary to delegate to the invariant and the super class.
7. Before implementing the code generation method, we should run our unit test to see that it now passes
 - a. Add more tests, including ones for code that should NOT compile.

5. Code Generation

- a. Implement the code generation for `JmlWhileStatement`.
 - i. Note: This step is left deliberately less detailed to get you to think about the problem on your own. A basic understanding of programming the JVM in bytecode is necessary.
 - ii. Questions:
 - How does the bytecode for a simple while loop look?
 - What is the control flow, specifically, how/when is the condition checked?
 - What are the blocks or pieces that the control flow can be broken into?
 - When should the invariant be checked?
 - iii. Add hooks to the necessary places in the super class's method.
 - iv. Override the hook method(s).