Tutorial 3: Slicing

CISC422/853

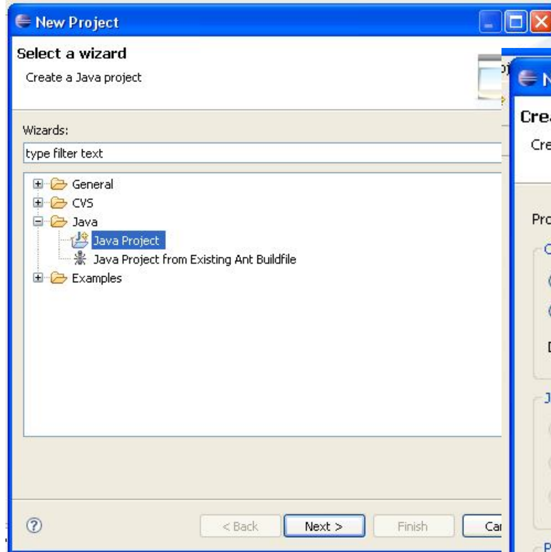
Scott Grant

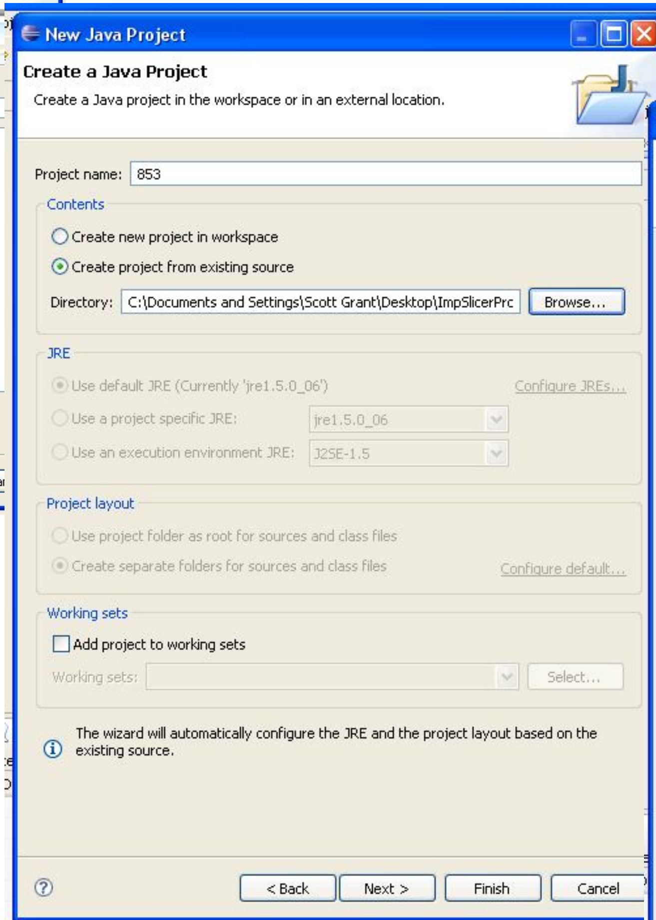
- Getting Started (Eclipse)
- Assignment Structure
- Advice for Assignment 4
- Debugging and Profiling in Eclipse
- Demonstration

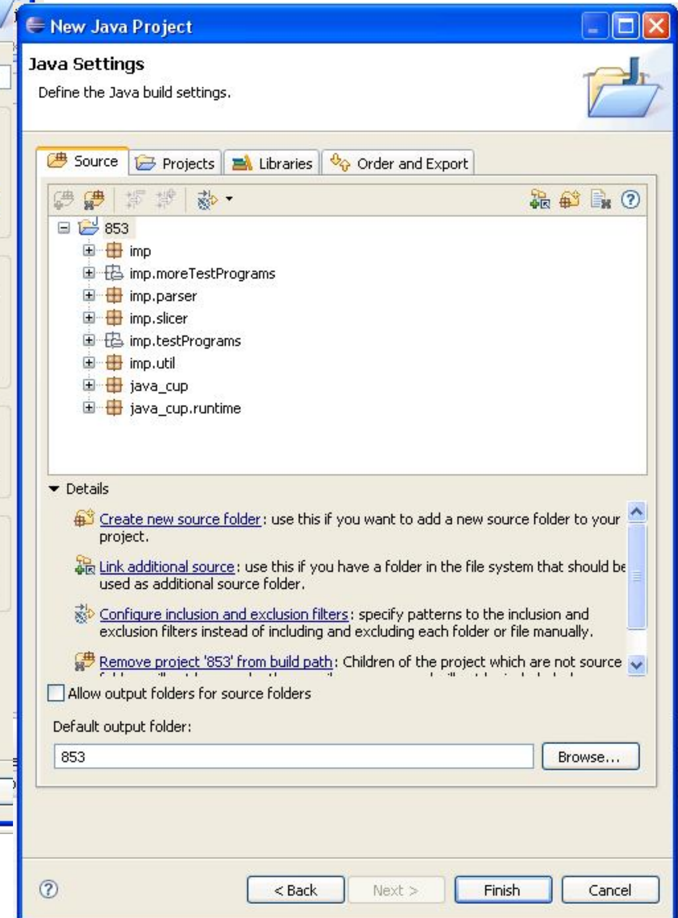
Getting Started (Eclipse)

- Download Eclipse, if you don't have it
  - If you downloaded Eclipse IDE for Java Developers (85 MB) for A1, you can use this
- Download a4CISC422853Winter2009.zip
  - Contains the Java source that you will be extending, and a set of IMP programs that you can use to test your solution
- In Eclipse, create a new Java Project
  - Import the files from the 422/853 zip archive

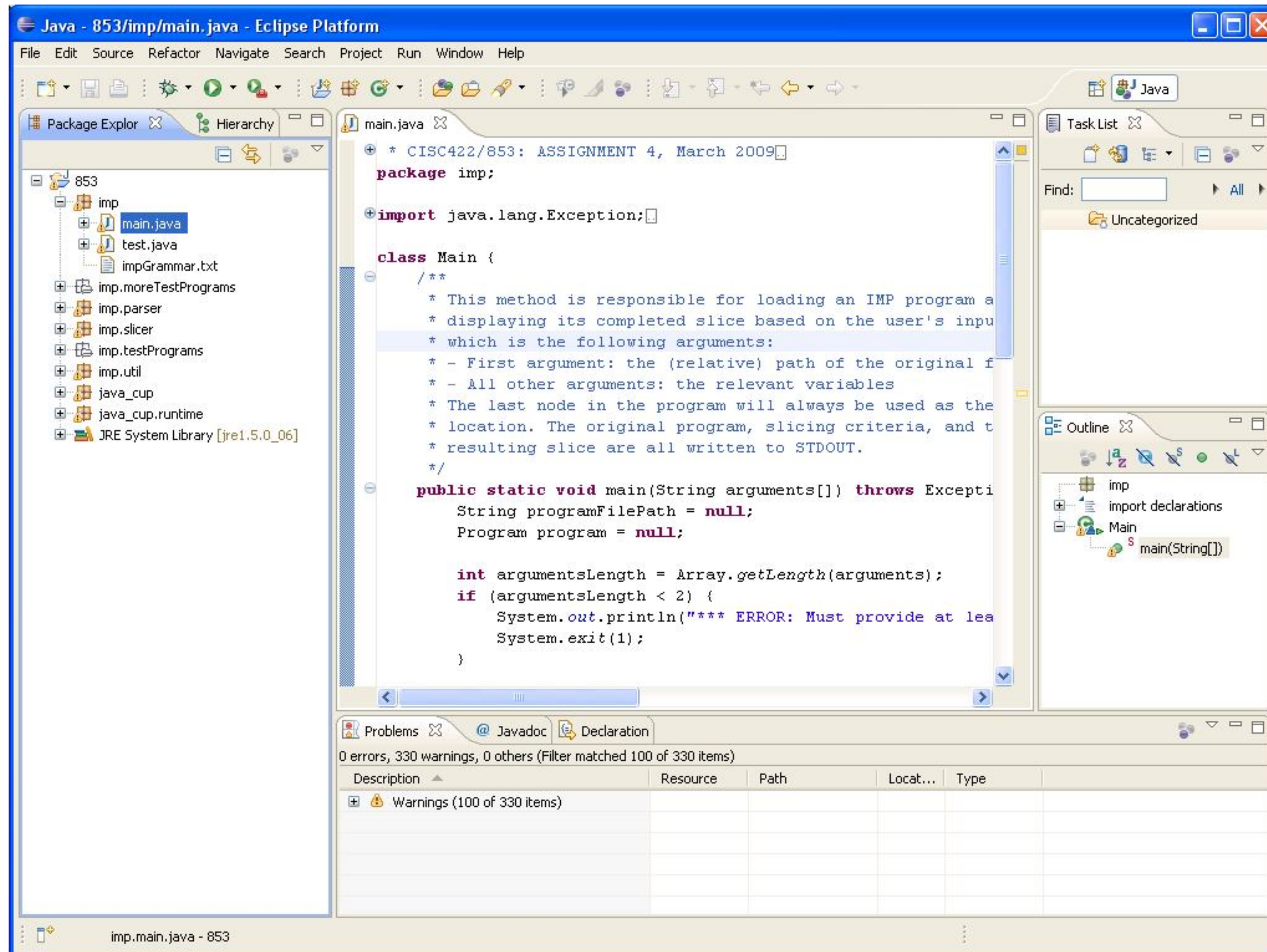
Getting Started (Eclipse)







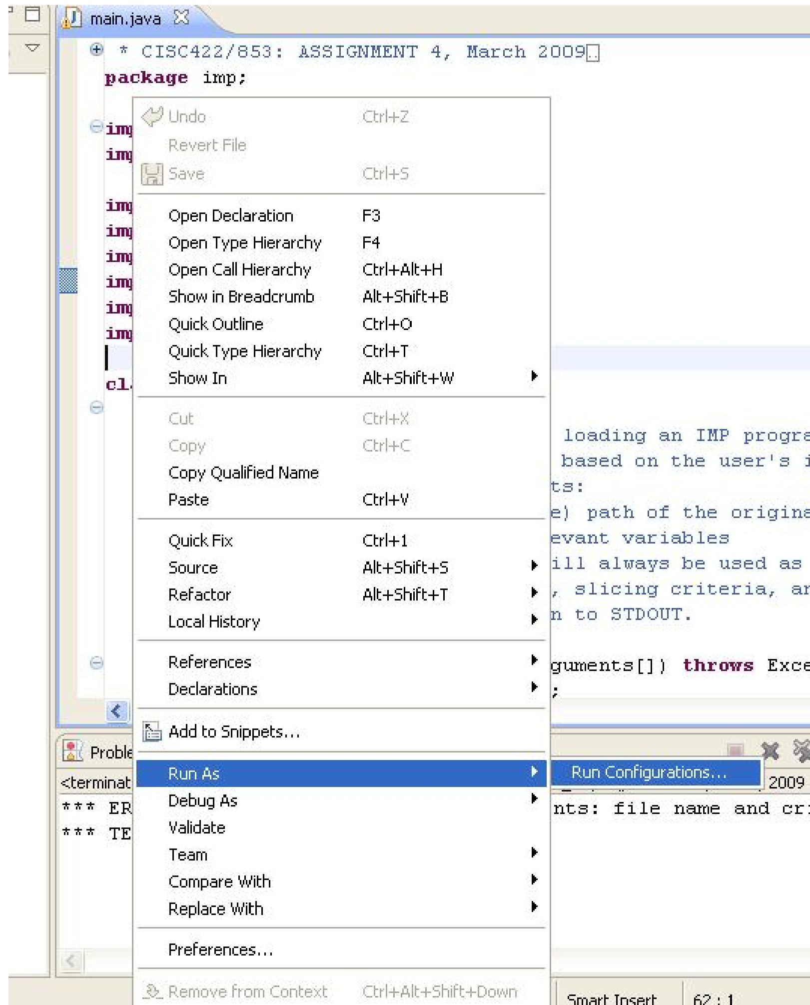
Getting Started (Eclipse)

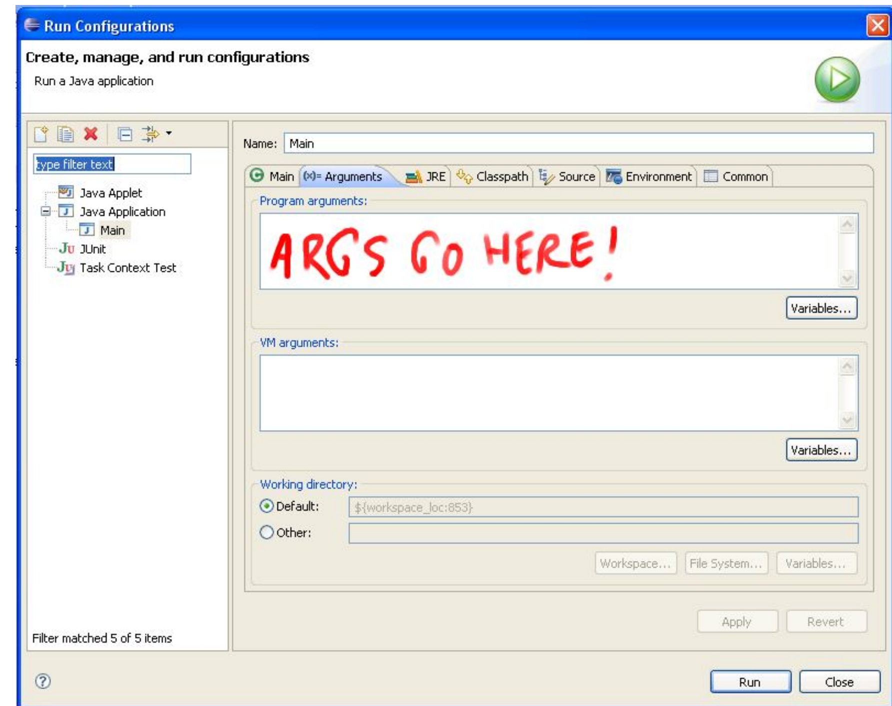


Getting Started (Eclipse)

- To verify that things are working:
  - Declare the command-line parameters to tell the slicer which file to use as input
  - Open /imp/main.java and right-click on the source window
  - Choose **Run As -> Run Configurations**
  - Run as a Java Application, and select the **Arguments** tab
  - The **Program arguments** box is where you will tell the slicer which file to process

Getting Started (Eclipse)





Getting Started (Eclipse)

- Try with a sample IMP program:
  - imp/testPrograms/p1.imp x

```
Reading Imp program from file imp/testPrograms/p1.imp

*** ORIGINAL PROGRAM ***

PROGRAM p1;
VAR
x : INT;
y : INT;
z : INT
0: BEGIN
1: x := 1;
2: y := 2;
3: PRINT((x+2));
4: x := 3;
5: z := (x+1)
6: END
```

```
*** SLICING CRITERIA ***

Location: 6: END
Variables: [x]

*** SLICED PROGRAM (WITHOUT VARIABLE DECLARATIONS) ***

0: BEGIN
6: END
```

Assignment Structure

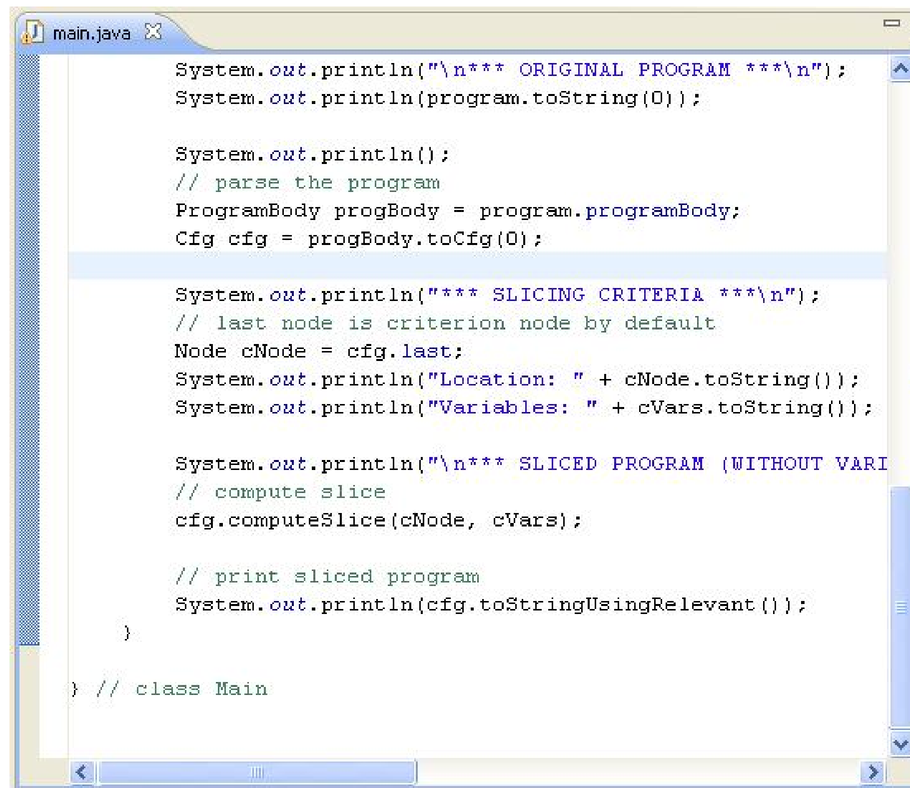
- What is all of this code doing?!
  - Technically, you only need to modify code in imp.slicer
    - Wait, that's not all, where are you going? Come back! It's interesting!
  - IMP has a parser generated from an LALR parser generator called CUP
    - You will have an Abstract Syntax Tree and a Control Flow Graph computed from the input program, and will use those to do your slicing

Assignment Structure

- What is all of this code doing?!
  - You aren't required to understand the parser, but it is very interesting (honestly, not just TA-speak)
  - If you want to "skim" compiler tech, and help dominate the assignment to boot, make sure you the understand the CFG, and pay close attention during the debugging part of this tutorial!

Assignment Structure

- Where do I begin?
  - One suggestion would be main.java



Assignment Structure

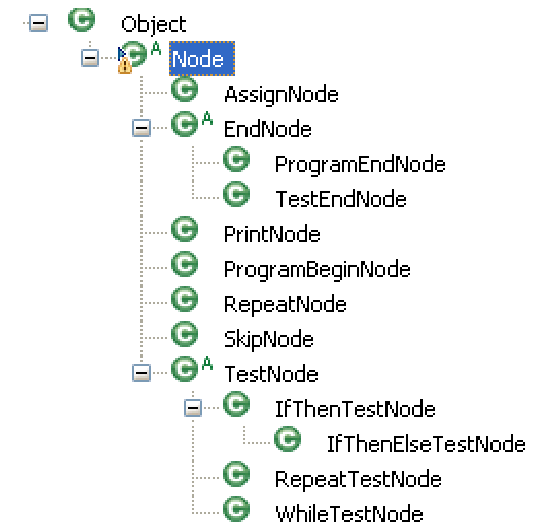
- cfg.computeSlice(cNode, cVars);
  - In main.java, determines the program slice
  - cNode is the current node in the Control Flow Graph
    - At first, this is the last node in the program
    - Node cNode = cfg.last;
  - cVars is the set of variables you list on the command line to compute the slice against
    - cVars.add("x");
    - if (cVars.contains("x")) { ... }

Assignment Structure

- cfg.computeSlice(cNode, cVars);
  - So cNode is the last node in the program's CFG, and cVars is the list of variables you want to compute the slice for
  - You will work backwards from cfg.last, passing information about the relevance of the variables
  - How? We'll see in a second, but first, what are Node objects?

Assignment Structure

- What is a Node object?
  - Each instantiation of a Node object represents a node in the CFG
  - Each Node instance has information that you can use
    - dRVars (directly relevant variables)
    - dRVarsChanged (help other Nodes)
    - isRelevant (relevant when true)
    - prevs and nexts (transitions)

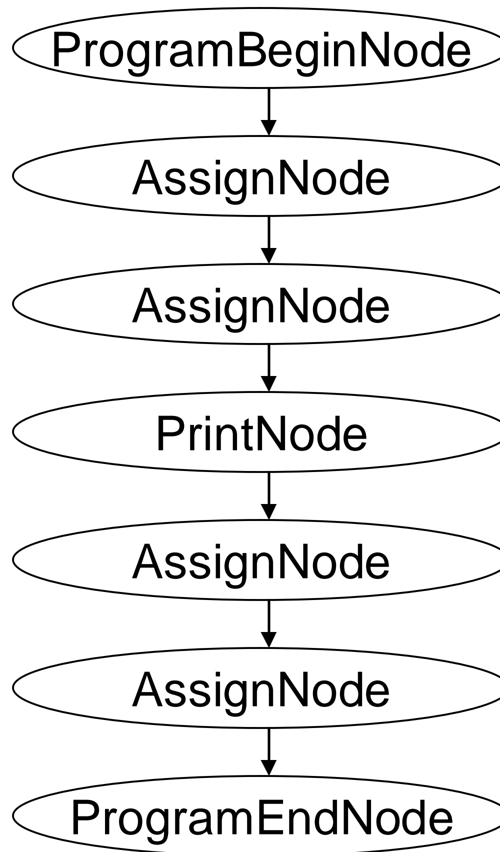


Assignment Structure

- What is a Node object?
  - A Node object roughly corresponds to a statement in your source program
    - There aren't nodes for variable x or variable y, there are nodes that identify assignment statements, or repeat loops
  - For the purposes of this assignment, Node objects are places where variables can become relevant to a slice

Assignment Structure

```
PROGRAM p1;
VAR
            x : INT;
            y : INT;
            z : INT
0: BEGIN
1:          x := 1;
2:          y := 2;
3:          PRINT((x+2));
4:          x := 3;
5:          z := (x+1)
6: END
```



This is an abstracted view of the cfg object that you'll have available.

Each prevs and nexts reference in a Node object is a Vector, so what are the elements of the prevs and nexts object for these?

Assignment Structure

Name	Value
cNode	ProgramEndNode (id=38)
dRVars	VarIdSet (id=94)
dRVarsChanged	false
indentLevel	0
isRelevant	false
loc	Loc (id=61)
nexts	Vector<E> (id=95)
capacityIncrement	0
elementCount	0
elementData	Object[10] (id=97)
modCount	0
prevs	Vector<E> (id=96)
capacityIncrement	0
elementCount	1
elementData	Object[10] (id=98)
[0]	AssignNode (id=99)
[1]	null
[2]	null
[3]	null
[4]	null
[5]	null
[6]	null
[7]	null
[8]	null
[9]	null
modCount	1
programBeginNode	ProgramBeginNode (id=44)

We'll explain this specifically in the demonstration, but here is a visual representation of cNode for the sample program imp/testPrograms/p1.imp.

nexts is an empty Vector, and prevs contains a single element to the AssignNode that precedes it in memory.

You can see other important variables here, like dRVars, dRVarsChanged, and isRelevant.

Name	Value
cVars	VarIdSet (id=25)

[x]

Name	Value
cNode	ProgramEndNode (id=38)
dRVars	VarIdSet (id=94)

[]

Assignment Structure

- What is dRVars?
  - A HashSet object in Java
  - Contains a set of String values corresponding to the relevant variables at this point in the slice
    - If x is relevant, then dRVars.contains("x") is true
  - This is important for passing information to earlier Node objects

Assignment Structure

- VarIdSet class definition
  - You can extend this if you feel some methods might help you with your slice

```
package imp.util;

import java.util.HashSet;

/* Implementation of a set containing the strings (id) inside Var objects.
 * Used to store the directly relevant variables.
 * Fill in this class as needed.
 */
public class VarIdSet extends HashSet {
}
```

Assignment Structure

- Adding entire dRVars objects?
  - This is just one example, you are not required to use it.
  - If you find your implementation uses lots of similar actions, you can extend the class

```
public void addVarIdSet(VarIdSet cVars) {
 Iterator<String> varIter = cVars.iterator();
 while (varIter.hasNext()) {
  this.add((String) varIter.next());
 }
}
```

Assignment Structure

- So, about that computeSlice method?
  - You will be mainly concerned with the computeDRVars method in Node objects under imp.slicer

```
// cfg.java
public void computeSlice(Node cNode, VarIdSet cVars) {
    cNode.computeDRVars(cNode, cVars);
}

// ProgramEndNode.java
public void computeDRVars(Node cNode, VarIdSet cVars) {
}
```

Assignment Structure

- If you run the code right now, what happens?
  - You compute the slice of your input program for the variables you specify on the command line
  - The computeSlice method begins at the ProgramEndNode point in the CFG, and calls computeDRVars to recursively derive the slice
  - ProgramEndNode has no code in computeDRVars, so it returns, and the slice is effectively empty

Assignment Structure

- Naive approach to get started
  - Pass relevant variables, look at previous nodes

```
// ProgramEndNode.java
public void computeDRVars(Node cNode, VarIdSet cVars) {
    this.dRVars.addVarIdSet(cVars);
    this.dRVarsChanged = true;
    this.isRelevant = true;

    for (int i=0; i<this.prevs.size(); i++) {
        Node prevNode = (Node) this.prevs.elementAt(i);
        if (!(prevNode instanceof ProgramBeginNode)) {
            prevNode.computeDRVars(this, this.dRVars);
        }
    }
}
```

Assignment Structure

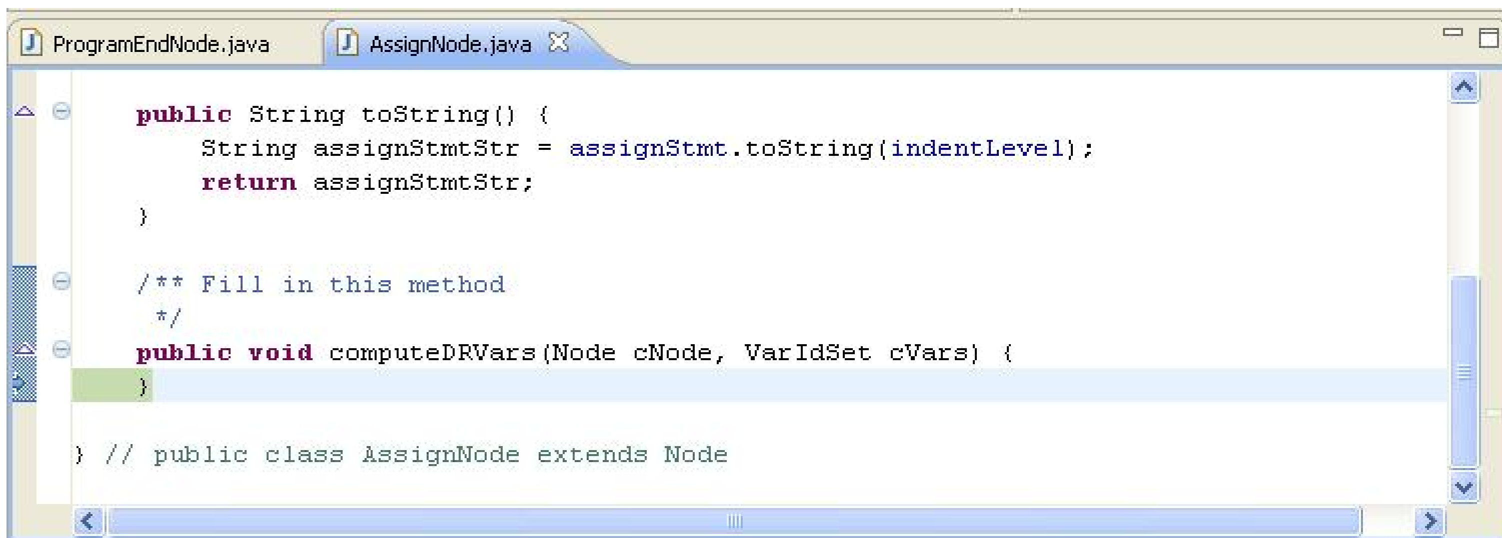
- What happens?
  - Same output, but very different internal result

```
Reading Imp program from file imp/testPrograms/p1.imp

*** ORIGINAL PROGRAM ***

PROGRAM p1;
VAR
x : INT;
y : INT;
z : INT
0: BEGIN
1: x := 1;
2: y := 2;
3: PRINT((x+2));
4: x := 3;
5: z := (x+1)
6: END
```

```
*** SLICING CRITERIA ***

Location: 6: END
Variables: [x]

*** SLICED PROGRAM (WITHOUT
    VARIABLE DECLARATIONS) ***

0: BEGIN
6: END
```

Assignment Structure

- Alright, we made it to AssignNode!
  - Of course, this is empty too. The saga continues..



Assignment Structure

- If you have questions about this process, we can cover them in the demonstration
  - (or of course, you can ask me now)
- This assignment relies on your ability to pass the correct relevant variables back through the CFG
  - Start with basic programs and work up to the complicated ones!

Advice for Assignment 4

- Start small
  - imp/testPrograms/p1.imp
  - What do you need to do with a PrintNode?
    - Can the print statement modify the relevant variables? What about SkipNode?
    - What should these computeDRVars methods look like?
  - Once you are comfortable with the AssignNode method, you will have a better idea of how the code is designed to work

Advice for Assignment 4

- Start early!
  - Okay, I say that with every assignment, but this one is important
  - This might actually feel like two assignments in one
    - The first assignment includes getting everything excluding loops working
    - The second comes when you realize how loops can complicate things
  - You'll probably want to save loops until the end

Advice for Assignment 4

- Don't assume the tests cover all cases
  - The test programs included with the code are pretty comprehensive, but you should try writing some IMP code to make sure your code does what you expect it will

Advice for Assignment 4

- Contact me or Juergen if you have questions
  - We want to help out, and if you give yourself enough time, we can get you on the right path
  - There are many ways to solve this problem
  - If you find things aren't working out, back up and revisit some earlier examples to get things working again

Debugging and Profiling in Eclipse

- You don't have to use Eclipse
  - If you're using another Java IDE (or just the command-line), there are other ways to debug - send me an email if you'd like some help
- If you use Eclipse, this can really help
  - Debugging isn't commonly taught in university curriculum
  - If you're going to get an industry job after school, debugging experience is really valuable

Debugging and Profiling in Eclipse

- What do I get out of it?
  - Normally when you run a piece of code, you don't have access to the line-by-line state of the variables
  - You can use print methods to get some information, but without debugging the code, you're extremely restricted in the information you can get
  - How would you see the entire CFG data structure as it exists in memory using a print statement?

Debugging and Profiling in Eclipse

Debug - 853/imp/main.java - Eclipse SDK

File Edit Source Refactor Navigate Search Project Run Window Help

Debug | Java

Debug

Main [Java Application]
imp.Main at localhost:1439
Thread [main] (Suspended (breakpoint at line 115 in Main))
Main.main(String[]) line: 115
C:\Program Files\Java\jre1.6.0_04\bin\javaw.exe (Mar 25, 2009 1:00:49 PM)

Variables | Breakpoints

Name	Value
cVars	VarIdSet (id=16)
progBody	ProgramBody (id=34)
cfg	Cfg (id=36)
first	ProgramBeginNode (id=54)
dRVars	VarIdSet (id=62)
dRVarsChange	false
indentLevel	0
isRelevant	false
loc	Loc (id=63)
nexts	Vector<E> (id=64)
prevs	Vector<E> (id=65)
last	ProgramEndNode (id=38)

```
0: BEGIN
1:  x := 1;
```

main.java

```
        System.out.println("\n*** SLICED PROGRAM (WITHOUT VARIABLE DECLARATIONS) ***\n
        // compute slice
        cfg.computeSlice(cNode, cVars);

        // print sliced program
        System.out.println(cfg.toStringUsingRelevant());
    }
```

Outline

imp
import declarations
Main
main(String[])

Console | Tasks

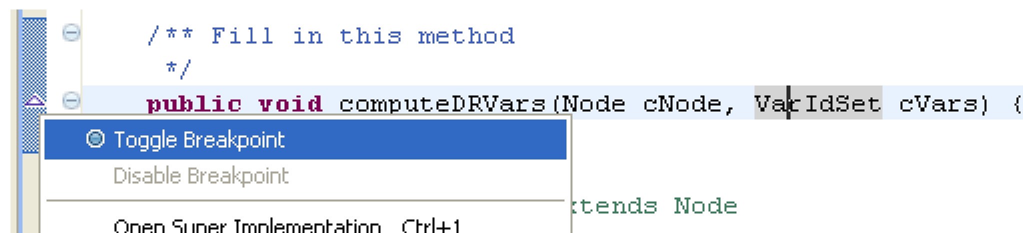
Main [Java Application] C:\Program Files\Java\jre1.6.0_04\bin\javaw.exe (Mar 25, 2009 1:00:49 PM)

```
Location: 6: END
Variables: [x]

*** SLICED PROGRAM (WITHOUT VARIABLE DECLARATIONS) ***
```

1 : 3

Debugging and Profiling in Eclipse

- If you want to examine specific parts of your program, use breakpoints
  - Set a breakpoint by either right-clicking on the left side of the source window, choosing Run -> Toggle Breakpoint, or pressing Ctrl-Shift-B
  - Make sure you choose "Debug" (F11) instead of just "Run" when you execute your code!



Debugging and Profiling in Eclipse

Resume/Stop/Step Into/Step Over/etc.

Current Location

Variable Watch

Code Window

Console Output

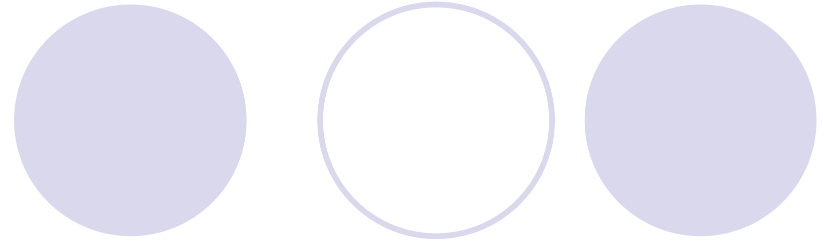
Debugging and Profiling in Eclipse

- Controlling code execution
  - Step Into (F5): Follow the trace into the current method, if possible
    - If we set a breakpoint at cfg.computeSlice and step into the code here, we retain control of execution and proceed inside the computeSlice method itself
  - Step Over (F6): Execute the current statement, and continue debugging on the next one
    - We don't care about the internals of this statement, but don't want to give up control yet

Debugging and Profiling in Eclipse

- Controlling code execution
  - Step Return (F7): Jump out a single level, out of the current method
  - Resume (F8): Continue debugging, and only stop again if we hit another breakpoint
  - Terminate (Ctrl-F2): Halt execution
    - If you're doing a lot of debugging, don't let your old processes sit around at breakpoints! Terminate them if you're done with them.

Demonstration



- Let's take a look at some breakpoints