CSE 390B, Winter 2023

Building Academic Success Through Bottom-Up Computing

Debugging Strategies & Project 8 Overview

Strategies for Debugging Software, Project 8 Introduction with AST Node Examples and MicroJack Overview

Lecture Outline

- **Strategies for Debugging Software**
  - **Debugging Process and The Scientific Method**

- Project 8 Overview
  - Number Literal and Plus AST Node Examples
  - Overview of MicroJack and Its Gotchas

Sources and Acknowledgements

- This is a subset and an adaptation of a CSE 331 lecture
- If you have taken CSE 331, you have seen this before
  - Part of your task for Project 8
  - This subject is closely connected to metacognition
- If you haven't taken CSE 331, this is a helpful sneak peek
  - Debugging is an important topic in many CSE courses
- Acknowledgements: CSE 331 instructors, notably Michael D. Ernst, Hal Perkins, and more

Debugging Pre-discussion

- How often do you run into bugs when writing programs?
- What is your debugging process?
  - In other words, when you run into a bug, do you have strategies that you consistently use to find it?
  - For those who have taken 331, maybe think back to before you had the debugging lecture
- What debugging strategies have you come across?

A Bug's Life

- Software bug definitions:
  - Defect – mistake committed by a human
  - Error – incorrect computation
  - Failure – visible error: program violates its specification
- Debugging starts when a failure is observed
  - During testing
  - In the field
- Goal is to go from failure back to defect

Testing Versus Debugging

- Testing ≠ debugging
  - Test: reveals existence of problem (failure)
  - Debug: pinpoint location + cause of problem (defect)

- See CSE 331 for:
  - How to write code that has fewer bugs (so less debugging)
  - How to write code that is easier to test (so easier to reveal bugs)
  - How to make testing easier (so you do it more often)
  - How to write code that is easier to debug (so less time spent debugging)

- These are all incredibly valuable engineering skills

Last (Inevitable) Resort: Debugging

- Defects happen, people are imperfect
  - Industry average: 10 defects per 1000 lines of code(?)

- Defects happen that are not immediately localizable
  - Found during integration testing
  - Or reported by user

- Cost of an error increases by orders of magnitude during program lifecycle

Debugging Lifecycle

- Step 1: Clarify symptom (simplify input), create "minimal" test
- Step 2: Find and understand cause
- Step 3: Fix and understand why it works
- Step 4: Rerun all tests, old and new

The Debugging Process

- Step 1: Find small, repeatable test case that produces the failure
  - May take effort, but helps identify the defect and gives you a regression test
  - Do not start Step 2 until you have a simple repeatable test

- Step 2: Narrow down location and proximate cause
  - Loop: (a) Study the data (b) hypothesize (c) experiment
  - Experiments often involve changing the code
  - Do not start Step 3 until you understand the cause

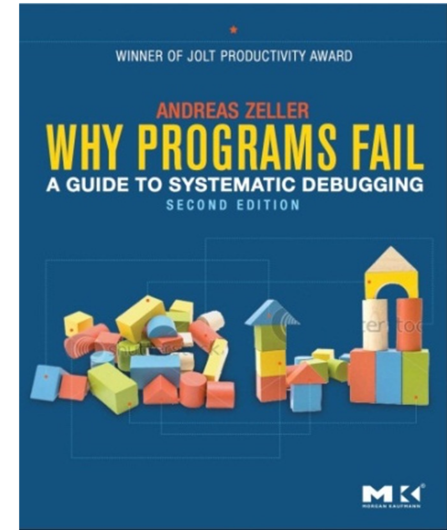
The Debugging Process

- Step 3: Fix the defect
  - Is it a simple typo, or a design flaw?
  - Does it occur elsewhere?

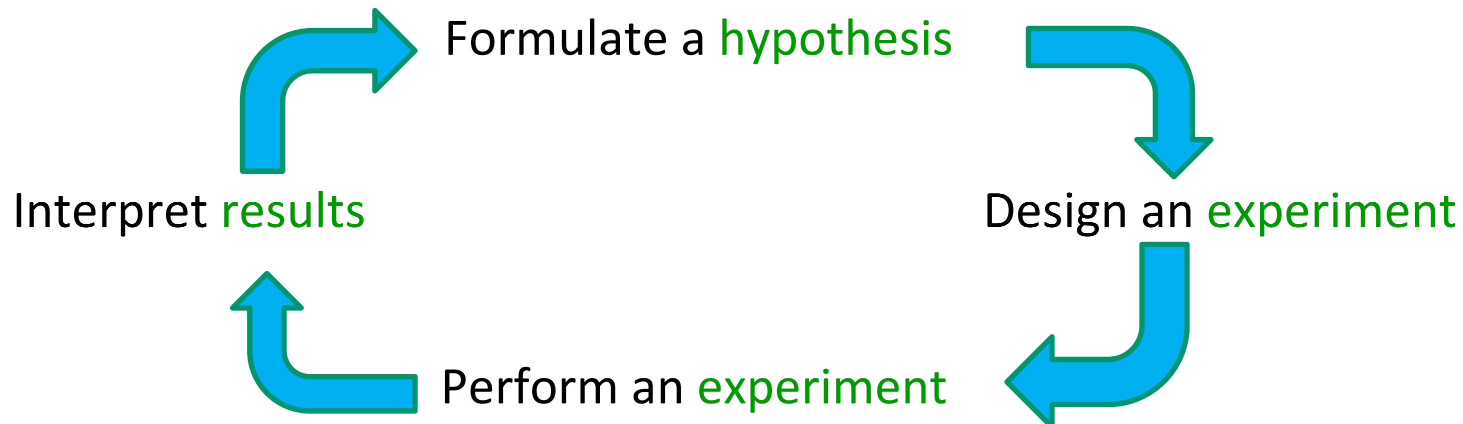
- Step 4: Add test case to regression suite
  - Is this failure fixed? Are any other new failures introduced?

Debugging and The Scientific Method

- Debugging should be systematic
  - Carefully decide what to do instead of flail
  - Keep a record of everything that you do
  - Don't get sucked into fruitless avenues



- Use an iterative scientific process:



Debugging Example

```
// returns true iff sub is a substring of full
// (i.e., iff there exists A,B such that full=A+sub+B)
boolean contains(String full, String sub);
```

- User bug report: Cannot find string "**very happy**" in:

  "Fáilte, you are very welcome! Hi Seán! I am very very happy to see you all."

- Poor responses:
  - Notice accented characters, panic about not knowing about Unicode, begin unorganized web searches and inserting poorly understood library calls, etc.
  - Start tracing the execution of this example
- Better response: simplify or clarify the symptom

Reducing Absolute Input Size

- Find a simple test case by divide-and-conquer

- Pare test down:

```
Cannot find "very happy" within
   "Fáilte, you are very welcome! Hi Seán! I am
   very very     happy to see you all.”
    "I am very very happy to see you all.”
    "very very happy"
Can find "very happy" within
      "very happy"
Cannot find "ab" within "aab"
```

Reducing Relative Input Size

- Can you find two almost identical test cases where one gives the correct answer and the other does not?

```
Cannot find "very happy" within
      "I am very very happy to see you all."

Can find "very happy" within
      "I am very happy to see you all.”
```

General Strategy: Simplify

- In general: Find simplest input that will provoke failure
  - Usually not the input that revealed existence of the defect

- Start with data that revealed the defect
  - Keep paring it down (“binary search” can help)
  - Often leads directly to an understanding of the cause

- When not dealing with simple method calls:
  - The “test input” is the set of steps that reliably trigger the failure
  - Same basic idea

Localizing a Defect

- Take advantage of modularity
  - Start with everything, take away pieces until failure goes away
  - Start with nothing, add pieces back in until failure appears

- Take advantage of modular reasoning
  - Trace through program, viewing intermediate results

- Binary search speeds up the process
  - Error happens somewhere between first and last statement
  - Do binary search on that ordered set of statements

Binary Search on Buggy Code

```
public class MotionDetector {
    private boolean first = true;
    private Matrix prev = new Matrix();

    public Point apply(Matrix current) {
        if (first) {
            prev = current;
        }
        Matrix motion = new Matrix();
        getDifference(prev,current,motion);
        applyThreshold(motion,motion,10);
        labelImage(motion,motion);
        Hist hist = getHistogram(motion);
        int top = hist.getMostFrequent();
        applyThreshold(motion,motion,top,top);
        Point result = getCentroid(motion);
        prev.copy(current);
        return result;
    }
}
```

no problem yet

Check intermediate result at half-way point

problem exists

Binary Search on Buggy Code

```
public class MotionDetector {
    private boolean first = true;
    private Matrix prev = new Matrix();

    public Point apply(Matrix current) {
        if (first) {
            prev = current;
        }
        Matrix motion = new Matrix();
        getDifference(prev,current,motion);
        applyThreshold(motion,motion,10);
        labelImage(motion,motion);
        Hist hist = getHistogram(motion);
        int top = hist.getMostFrequent();
        applyThreshold(motion,motion,top,top);
        Point result = getCentroid(motion);
        prev.copy(current);
        return result;
    }
}
```

no problem yet

Check intermediate result at half-way point

problem exists

Detecting Bugs in the Real World

- Real Systems
  - Large and complex
  - Collection of modules, written by multiple people
  - Complex input
  - Many external interactions
  - Nondeterministic
- Replication can be an issue
  - Infrequent failure
  - Instrumentation eliminates the failure
  - No printf or debugger
- Errors cross abstraction barriers
- Large time lag from corruption (error) to detection (failure)

Heisenbugs

- In a sequential, deterministic program, failure is repeatable
- But the real world is not that nice…
  - Continuous input/environment changes
  - Timing dependencies
  - Concurrency and parallelism
- Failure occurs randomly
  - Depends on results of random-number generation
  - Hash tables behave differently when program is rerun
- Bugs hard to reproduce when:
  - Use of debugger or assertions makes failure goes away
    - Due to timing or assertions having side-effects
  - Only happens when under heavy load and once in a while

Logging Events

- Log (record) events during execution as program runs (at full speed)

- Examine logs to help reconstruct the past
  - Particularly on failing runs
  - And/or compare failing and non-failing runs

- But don't spend too much time manually reading enormous, confusing logs

More Tricks for Hard Bugs

- Rebuild system from scratch, or restart / reboot
  - Find the bug in your build system or persistent data structures
- Explain the problem to a friend (or to a rubber duck)
- Make sure it is a bug
  - Program may be working correctly and you don't realize it
- Face reality
  - Debug reality (actual evidence), not what you think is true
- And things we already know:
  - Minimize input required to exercise bug (exhibit failure)
  - Add more checks to the program
  - Add more logging

Where is the Defect?

- The defect is not where you think it is
  - Ask yourself where it cannot be; explain why
  - Self-psychology: look forward to being wrong!
- Look for simple easy-to-overlook mistakes first, e.g.,
  - Reversed order of arguments
  - Spelling of identifiers
  - Same object vs. equal: a == b versus a.equals(b)
  - Uninitialized data / variables
  - Deep vs. shallow copy
- Make sure that you have correct source code!
  - Check out fresh copy from repository; recompile everything
  - Does a syntax error break the build? (it should!)

When Debugging Gets Tough

- Reconsider assumptions
  - Debug the code, not the comments
    - Ensure that comments and specs describe the code
- Start documenting your system
  - Gives a fresh angle, and highlights area of confusion
- Ask for help
  - We all develop blind spots
  - Explaining the problem often helps (even to rubber duck)
- Walk away
  - Trade latency for efficiency – sleep!
  - One good reason to start early

Key Debugging Concepts

- Testing and debugging are different
  - Testing reveals existence of failures
  - Debugging pinpoints location of defects
- Debugging should be a systematic process
  - Use the scientific method
- Understand the source of defects
  - To find similar ones and prevent them in the future
- Learn from the debugging process
  - It’s inevitable and you have some control over how you approach the frustration

Debugging Post-discussion

- What is one useful thing you learned about debugging in this lecture?
- How might you change your debugging process after learning about these debugging strategies?
- How might you use it on debugging Project 8? For other projects?

Lecture Outline

- Strategies for Debugging Software
  - Debugging Process and The Scientific Method
- **Project 8 Overview**
  - **Number Literal and Plus AST Node Examples**
  - **Overview of MicroJack and Its Gotchas**

Project 8 Overview

- You will be given starter code for a compiler that reads a micro version of Jack and spits out Hack

- The Scanner & Parser are working
  - Task A: read through comments to understand what's going on

- The Code Generation is buggy and half-finished
  - Task B: find the bugs by practicing deliberate debugging strategies (e.g., step through generated Hack code using CPUEmulator)
  - Task C: Complete the implementation of the compiler

Project 8: MicroJack

- Stripped-down version of Jack language
  - More manageable but enough features to be interesting
- Available features:
  - Types: **`int`** and **`int[]`**, **`var`**
  - Structures: **`if`**, **`while`**, **`+`**, **`-`**, **`==`**, **`!=`**
- Missing features:
  - Functions, function calls, classes, objects, strings, for loops, array bounds checking, etc.

Any number of variable declarations

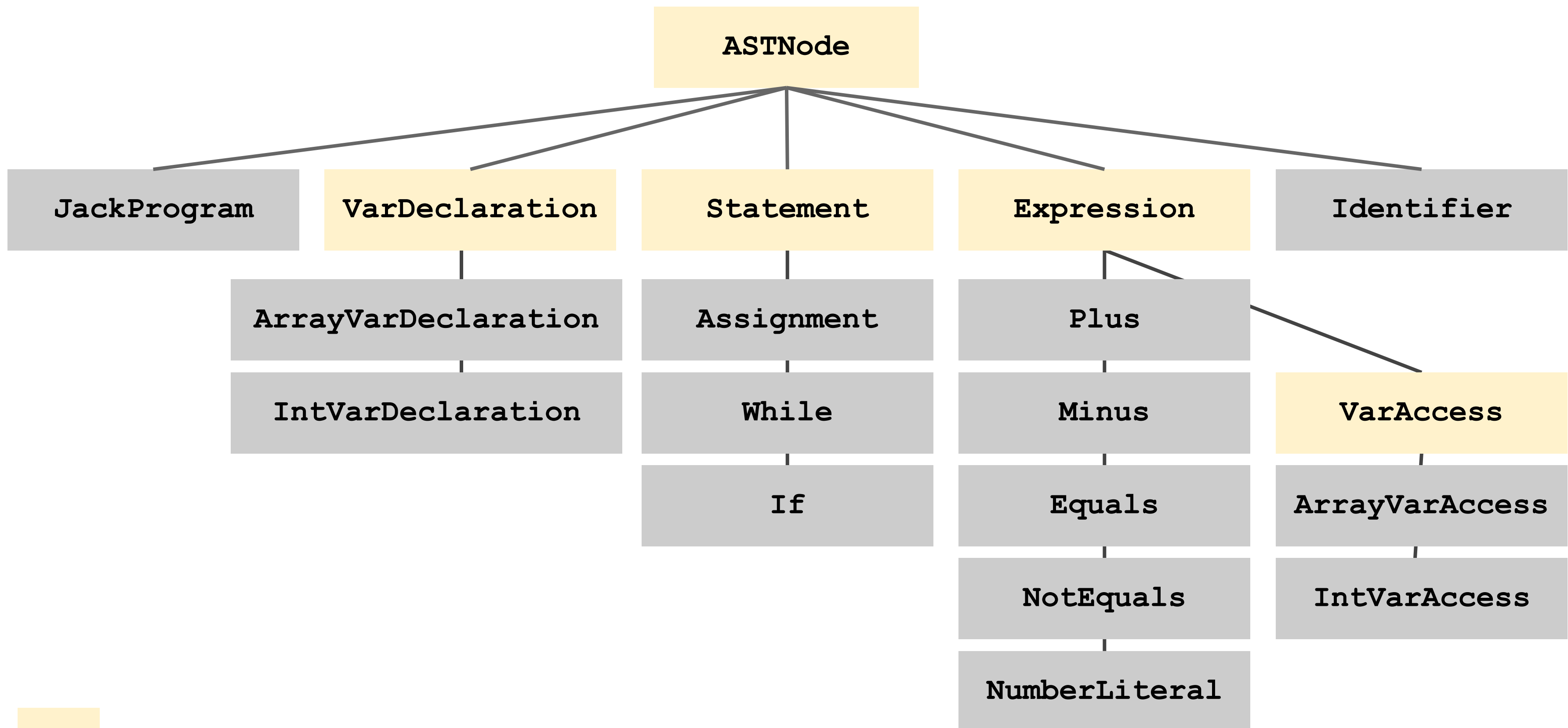
Basic.jack

```
var int a, b[1], c;
var int d[10], e;

let a = 1;
let b[0] = 1;
let n = 9;
while (n != 0) {
  let d[n] = a;
  let n = n - 1;
}
let screen[100] = d[0];
```

Then any number of statements

Project 8: The AST Nodes

- You are provided with all AST Node classes needed
  - All your code will be implemented within these classes



Project 8: Generating Code

- Each AST node has a printASM method that should print out Hack instructions to System.out (and recursively call printASM on children)
  - You're provided with **`instr("@R0")`** and **`label("LOOP")`** convenience functions
  - Each can take a comment as a second argument — highly recommended!

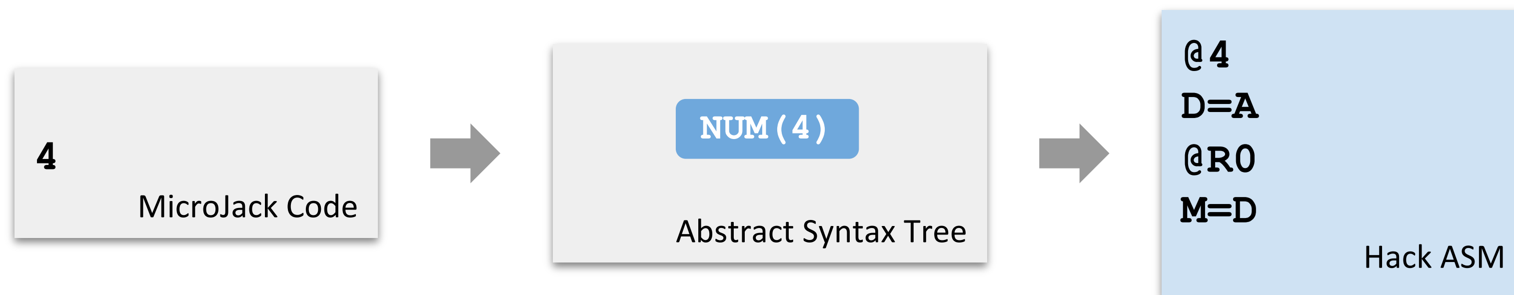
```
public class If extends Statement {
    public Expression condition;
    public List<Statement> statements;

    @Override
    public void printASM(symbolTable) {
        condition.printASM(symbolTable);
        instr("@R0", "Get cond result");
        instr("D=M");

        ...
    }
}
```

Project 8 Overview

- Step 1: Read comments provided in the starter code
- Step 2: Implement **`NumberLiteral.java`** (~4 lines)
- Step 3: Debug **`Plus.java`** (2 bugs)
- Step 4: Implement **`Minus.java`** (~13 lines, similar to **`Plus.java`**)
- Step 5: Implement **`NotEquals.java`** (~21 lines, similar to **`Equals.java`**)
- Step 6: Implement **`ArrayVarAccess.java`** (~3 lines)
- Step 7: Debug **`If.java`** (2 bugs)
- Step 8: Implement **`While.java`** (~14 lines)

Example: Number Literal (Step 1)

- Called a "literal" because it's a literal value embedded in the MicroJack code
  - Generated Hack Assembly should simply put that value in R0



Example: Number Literal (Step 1)

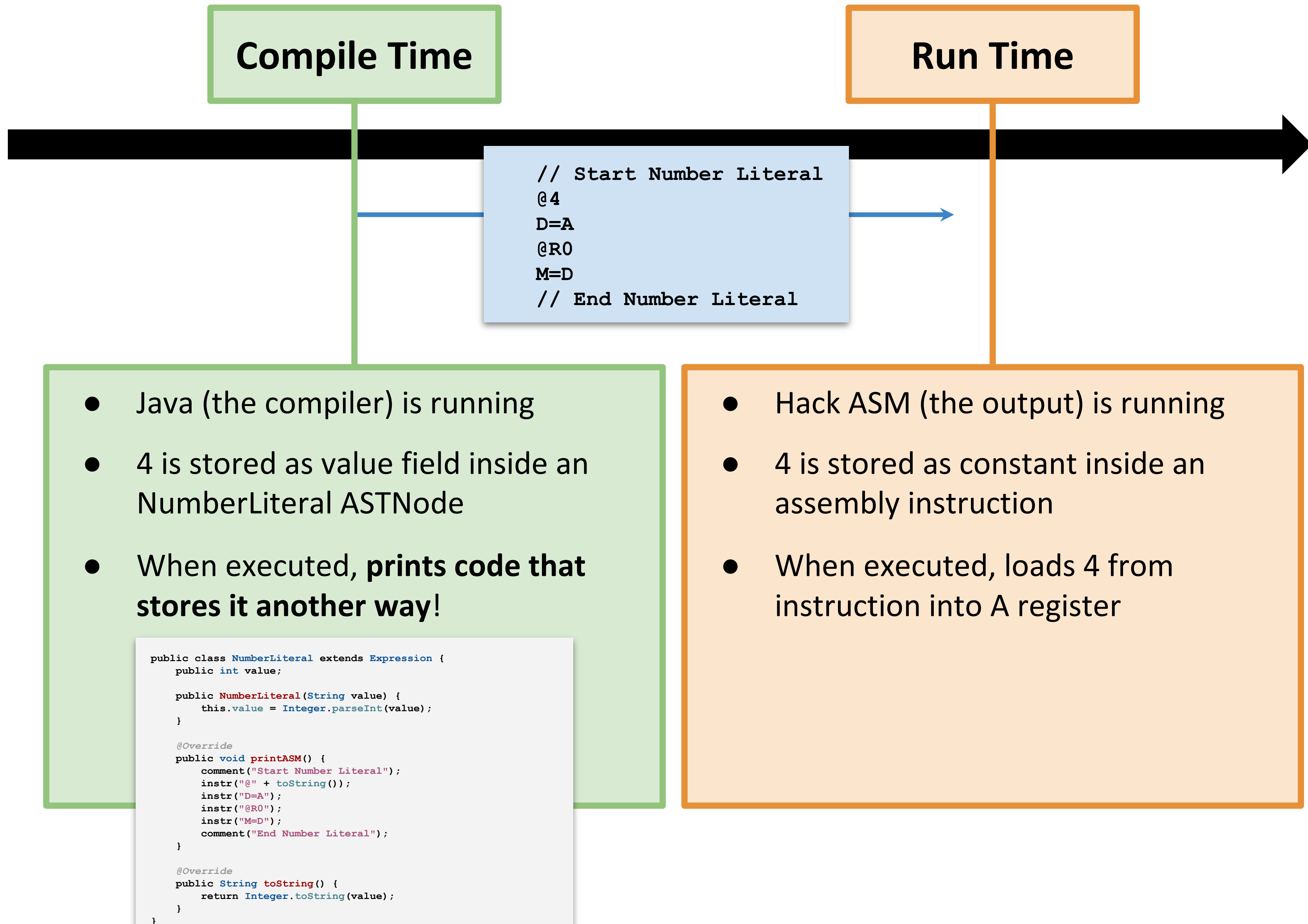
```
public class NumberLiteral extends Expression {
    public int value;        4

    public NumberLiteral(String value) {
        this.value = Integer.parseInt(value);
    }

    @Override
    public void printASM() {
        comment("Start Number Literal");  ➡  // Start Number Literal
        instr(           ?           );   ➡  @4
        instr("D=A");                     ➡  D=A
        instr("@R0");                     ➡  @R0
        instr("M=D");                     ➡  M=D
        comment("End Number Literal");    ➡  // End Number Literal
    }

    @Override
    public String toString() {
        return Integer.toString(value);
    }
}
```

Example: Number Literal (Step 1)

```
public class NumberLiteral extends Expression {
    public int value;    4

    public NumberLiteral(String value) {
        this.value = Integer.parseInt(value);
    }

    @Override
    public void printASM() {
        comment("Start Number Literal");
        instr("@" + toString());
        instr("D=A");
        instr("@R0");
        instr("M=D");
        comment("End Number Literal");
    }

    @Override
    public String toString() {
        return Integer.toString(value);
    }
}
```

```
// Start Number Literal
@4
D=A
@R0
M=D
// End Number Literal
```

Example: Number Literal (Step 1)



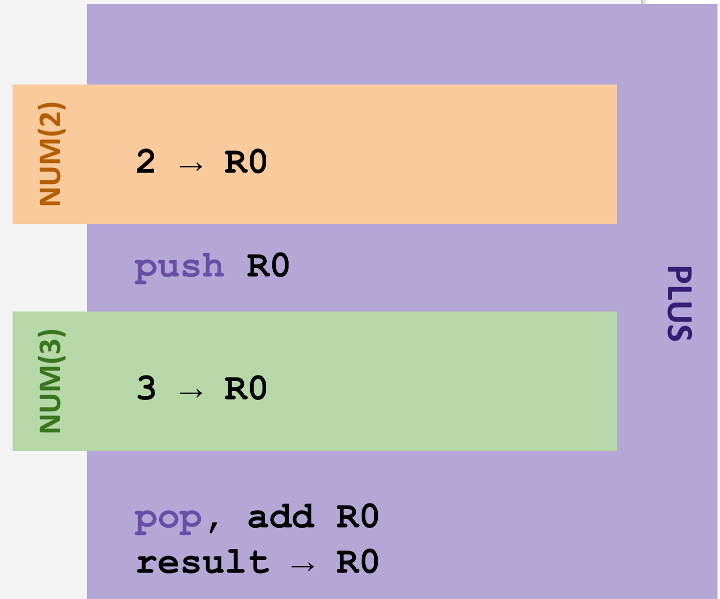
Example: Plus (Step 2)

```
public class Plus extends Expression {
    public Expression left;
    public Expression right;

    @Override
    public void printASM() {
        comment("Start Plus");
        left.printASM();
        instr("@R0");
        instr("D=M");
        right.printASM();
        push();
        instr("@R0");
        instr("D=M");
        instr("@R1");
        instr("A=M", "Get address of top of the stack");
        instr("D=D+A", "Perform the addition");
        instr("@R0");
        instr("M=D");
        pop();
        comment("End Plus");
    }
}
```

Example: Plus (Step 2)

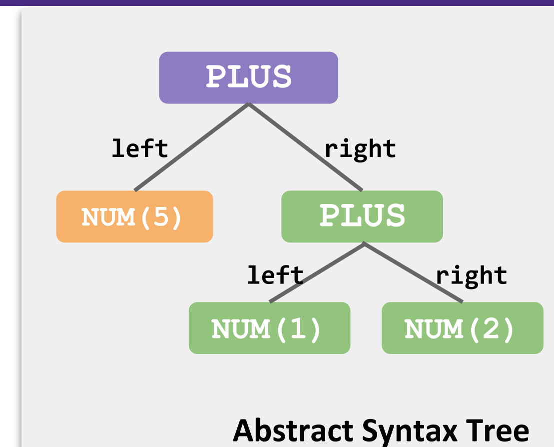
1 Structural Bug: Map to abstract diagram for Plus:

```
public class Plus extends Expression {
    public Expression left;
    public Expression right;

    @Override
    public void printASM() {
        comment("Start Plus");
        left.printASM();
        instr("@R0");
        instr("D=M");
        right.printASM();
        push();
        instr("@R0");
        instr("D=M");
        instr("@R1");
        instr("A=M", "Get address of top of the stack);
        instr("D=D+A", "Perform the addition");
        instr("@R0");
        instr("M=D");
        pop();
        comment("End Plus");
    }
}
```



1 Detail Bug: Step through generated code, Check state at each step

Code Generation: Example



- Solution: Store "saved" values in a stack
  - Not quite the same as "The Stack" or function call stack frames (but used for a similar reason)
- We'll keep a stack starting at memory address 1024
  - R1 is our *stack pointer*: always stores address of last used stack position
  - No built-in Hack push: manually copy to memory and increment R1

PLUS

NUM(2)

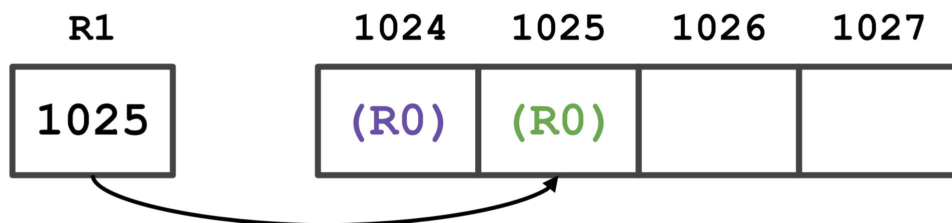
```
@5
D=A
@R0
M=D
```

```
// push R0 to slot 0
```

PLUS

```
@1
...
// push R0 to slot 1
...
// pop R0 from slot 1
@R0
MD=D+M
```

```
@R0
D=M
// pop R0 from slot 0
@R0
MD=D+M
```



MicroJack Gotchas

- Can't write a negative integer literal
  - Instead, use subtraction from zero: `0 - 1`
- All variable declarations must come before all regular statements
  - Why? Simplifies concept of a "defined" variable
- No defined operator precedence
  - If order matters for an operation, use parentheses

MicroJack Gotchas

- Arrays are just as you would expect
  - **`arr[index]`** just calculating an address: take address of **`arr`** variable and add index to it as an offset
  - No array bounds checking — you can run off the end of an array
- Booleans are 0 (false) and non-zero (true)

Project 8: Debugging Tips

- Try walking through the general `printASM` code to understand why each line is there
  - Add comments to the assembly as you go! Much easier to understand resulting file
- Find the smallest example you can
  - Provided tests get progressively more complex, but you may want to write your own tiny test case to isolate
  - `printASM` methods can get long fast—we've added comments so you can isolate to the section you're working on
- "Play Computer": as you step through the code, write down the state you expect after each instruction, then advance and see if the CPUEmulator agrees

Additional Project 8 Tips

- When debugging assembly, a good first step is to try understanding the code and adding comments to the assembly as you go
  - Much easier to understand resulting file

- A **`printDebug`** method has been implemented for you on all AST nodes
  - Use it to visualize exactly what the parser is giving you, but also as a basis for **`printASM`**
  - Both need to do processing on the current node and strategically recurse on its children

Additional Project 8 Tips

- Pushing and popping from the stack can be intimidating, but formulaic
  - Understand it once, copy and paste afterward
  - **`push()`** and **`pop()`** are already implemented for you

- We provide only a few MicroJack test files
  - We encourage you to write more of your own (think back to the debugging lecture)
  - Can use **`Sandbox.*`** to write more tests or create your own files

Project 8 Tools Practice

- Practice using the Project 8 tools — try the following:
  - Run **`git pull`** to pull the Project 8 starter code
  - Navigate to the **`src`** directory: **`cd src`**
  - Compile the Java source code of the compiler by running: **`javac $(find . -name "*.java")`**
  - Use your compiler to compile the Jack file for the **`OnlyVars.jack`** program: **`java compiler/Compiler compile ../test/OnlyVars.jack`**
  - Load and run **`OnlyVars.tst`** in the CPUEmulator

- The above steps were taken from the "How to Run Tests" portion of the specification
  - Can refer to this when needed as you work through the project

Post-Lecture 16 Reminders

- Next week: Operating Systems and Computer Networks!

- Project Reminders
  - **Project 7, Part I: Midterm Corrections due tonight (2/23) at 11:59pm (no late days may be used)**
  - Project 7, Part II: Professor Meeting Report due next Thursday (3/2) at 11:59pm
  - Project 8: Debugging & Implementing a Compiler released, due next Tuesday (3/7) at 11:59pm

- Eric has office hours after class in CSE2 153
  - Feel free to post your questions on the Ed board as well