

CSE 390B, Spring 2022

Building Academic Success Through Bottom-Up Computing

Debugging Strategies & Code Generation

Debugging Metacognitive Skill, Roadmap of Compilers, Code
Generation, Two-Tier Compilation

Mid-quarter Feedback

- ❖ Thanks for filling this out!

- ❖ Most helpful parts for learning in this course:
 - Student-TA 1:1 Meetings
 - Attending office hours
 - Lecture pre-readings

- ❖ Common factors that hinder learning in this course:
 - Time management (due dates, fast pace and keeping up with 390B)
 - Feeling a disconnect between lectures and projects
 - Project specifications not detailed enough

Lecture Outline

- ❖ **Debugging Metacognitive Skill**
 - **Debugging Process and Strategies, The Scientific Method**
- ❖ Roadmap of Compilers
 - Scanner, Parser, Type Checker, Optimizer, Code Generation
- ❖ Compilers: Code Generation
 - Generating Target Code from AST
- ❖ Two-Tier Compilation
 - Intermediate Programs and The Java Virtual Machine (JVM)

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 \neq 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

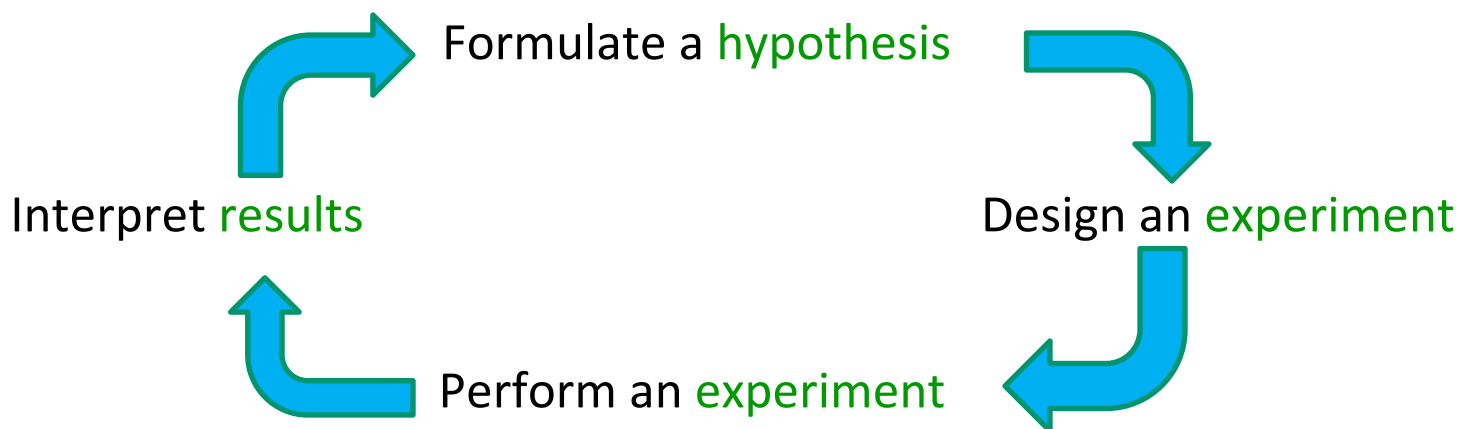
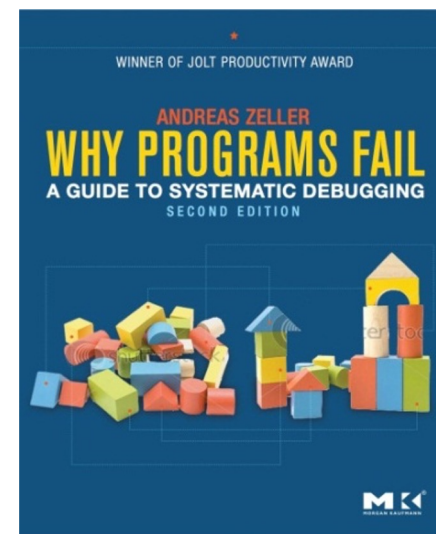
The Debugging Process

- ❖ 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, ...
 - 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

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 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

Lecture Outline

- ❖ Debugging Metacognitive Skill
 - Debugging Process and Strategies, The Scientific Method
- ❖ Roadmap of Compilers
 - Scanner, Parser, Type Checker, Optimizer, Code Generation
- ❖ Compilers: Code Generation
 - Generating Target Code from AST
- ❖ Two-Tier Compilation
 - Intermediate Programs and The Java Virtual Machine (JVM)



Vote at <https://pollev.com/cse390b>

Which of the following is NOT a useful property of the Abstract Syntax Tree (AST)?

- A. ASTs are a general, recursive structure that allows us describe any program**
- B. ASTs allow the compiler to extract relevant parts of a computer program**
- C. ASTs enable the compiler to perform type checking of the input program**
- D. ASTs perform optimizations in the intermediate representations of a program**
- E. We're lost...**

Software Overview

Compiler
(Project 8)

High-Level Language

- Java
- Python
- C/C++
- Jack

Compiler

Intermediate Language(s)

- Java Byte Code
- Jack VM Code

Compiler (VM Translator)

Assembly Language

- x86, x86-64
- ARM
- RISC-V
- HACK

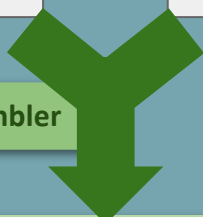
Operating System

- Windows
- Mac
- Unix/Linux
- Android
- Hack OS

Assembler

Machine Code

SOFTWARE



The Compiler: Implementation

```
public int fact(int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * fact(n - 1);  
    }  
}
```

High-Level Language

```
(fact)  
    @R0  
    M=M+1  
    @R1  
    D=A  
    @ifbranch  
    D;JEQ
```

Assembly Language

Scanner

Parser

Type
Checker

Optimizer

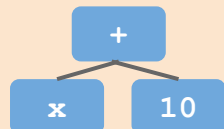
Code
Generator

Break string into
discrete **tokens**:

IF (ID(n)

== NUM(0) etc.

Arrange tokens into
syntax tree:



Verify the
syntax tree is
**semantically
correct**

Rearrange the
code to be
more efficient

Convert the syntax
tree to the **target
language**

Five-minute Break!

- ❖ Feel free to stand up, stretch, use the restroom, drink some water, review your notes, or ask questions
- ❖ We'll be back at: 3:30pm
- ❖ Research shows mid-lecture breaks reduce the decline of attention in the middle of lecture (Olmsted, 1999)

Lecture Outline

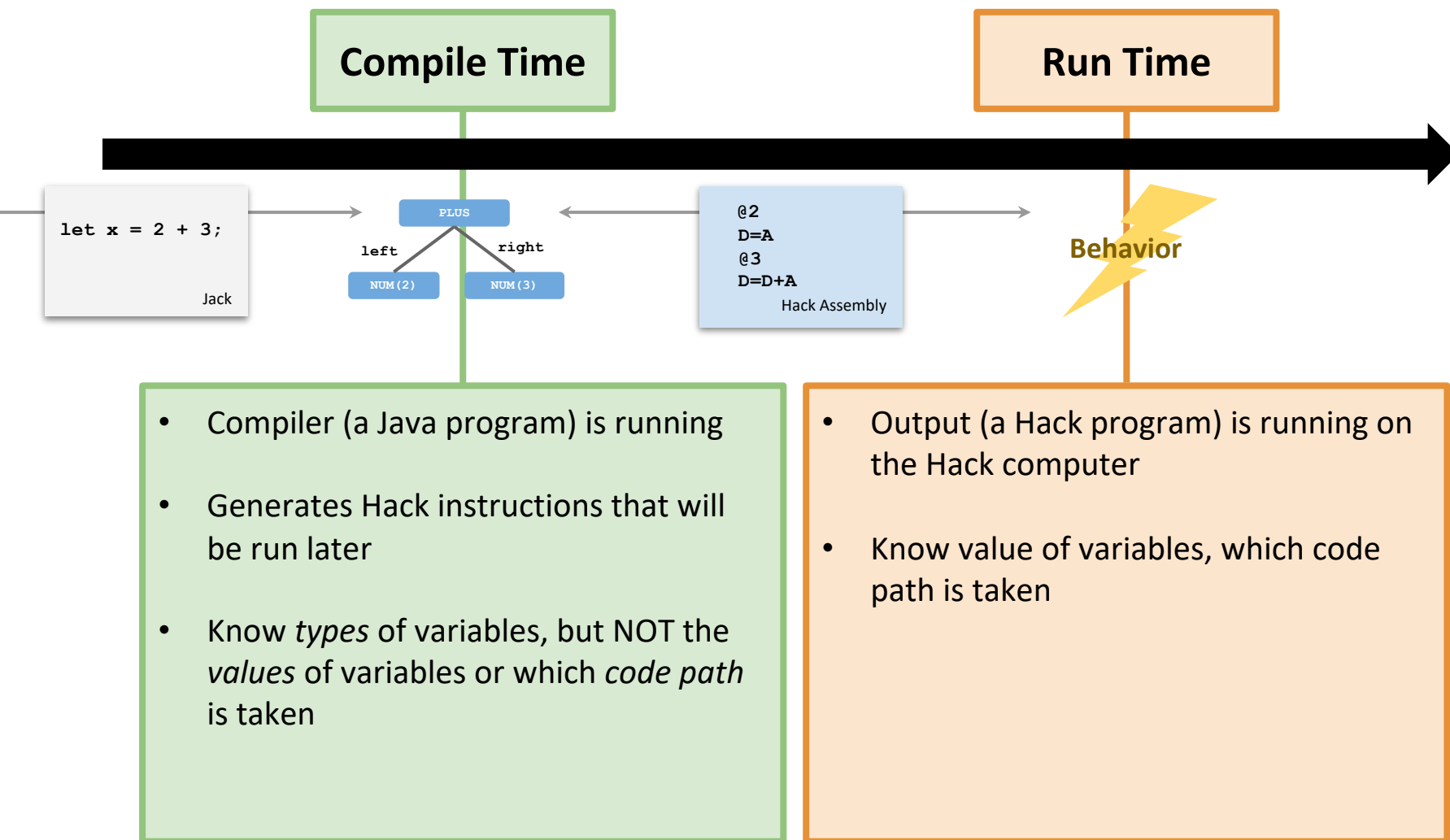
- ❖ Debugging Metacognitive Skill
 - Debugging Process and Strategies, The Scientific Method
- ❖ Roadmap of Compilers
 - Scanner, Parser, Type Checker, Optimizer, Code Generation
- ❖ **Compilers: Code Generation**
 - **Generating Target Code from AST**
- ❖ Two-Tier Compilation
 - Intermediate Programs and The Java Virtual Machine (JVM)

Code Generation: The Task



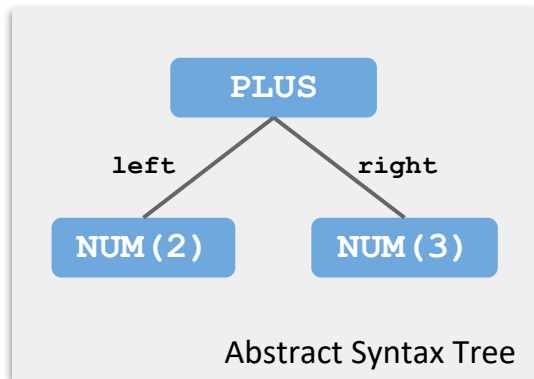
- ❖ Convert the AST into **target language code** that produces the same result
- ❖ Project 8 Goal: Produce **reliable**, not efficient, compiler
- ❖ The tricky bit: Do it automatically for all possible arrangements of code
 - To stay sane, we'll break the task down:
 - Generate code *for each node type* in the AST

Compile Time vs. Run Time



Code Generation: Example

- ❖ Here's how you, a brilliant human, would likely translate this syntax tree into Hack:



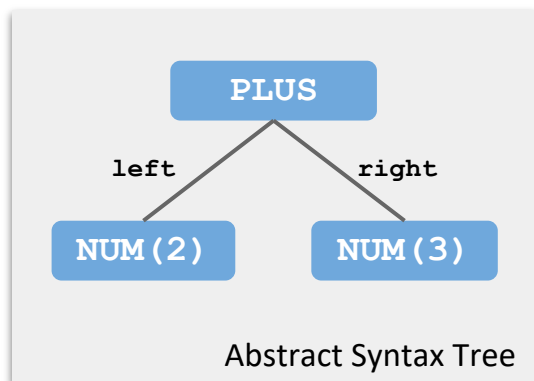
Human
(genius)

```
@2  
D=A  
@3  
D=D+A
```

Hack Assembly

Code Generation: Example

- ❖ Here's how you, a brilliant human, would likely translate this syntax tree into Hack:



Human
(genius)

```
@2
D=A
@3
D=D+A
```

Hack Assembly

Computer
(trying its
best)

```
@2
D=A
@R0
M=D
// save R0 somehow

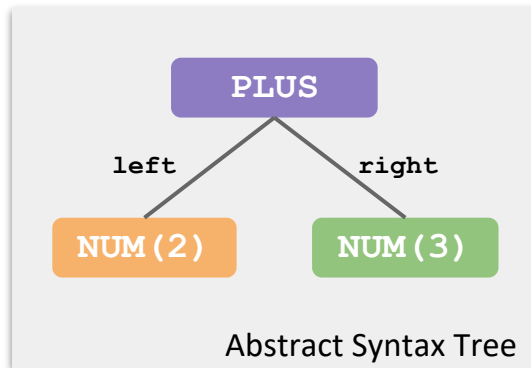
@3
D=A
@R0
M=D

@R0
D=M
// restore R0
@R0
MD=D+M
```

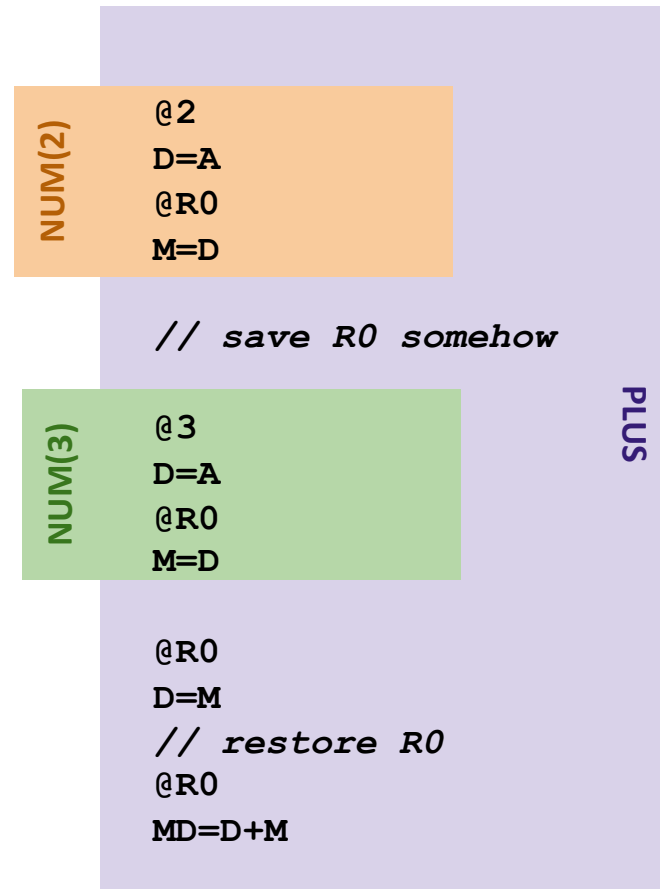
Hack Assembly

Code Generation: Example

- ❖ Why? Modularity: We can fit any expression in that slot, as long as **its result ends up in R0!**

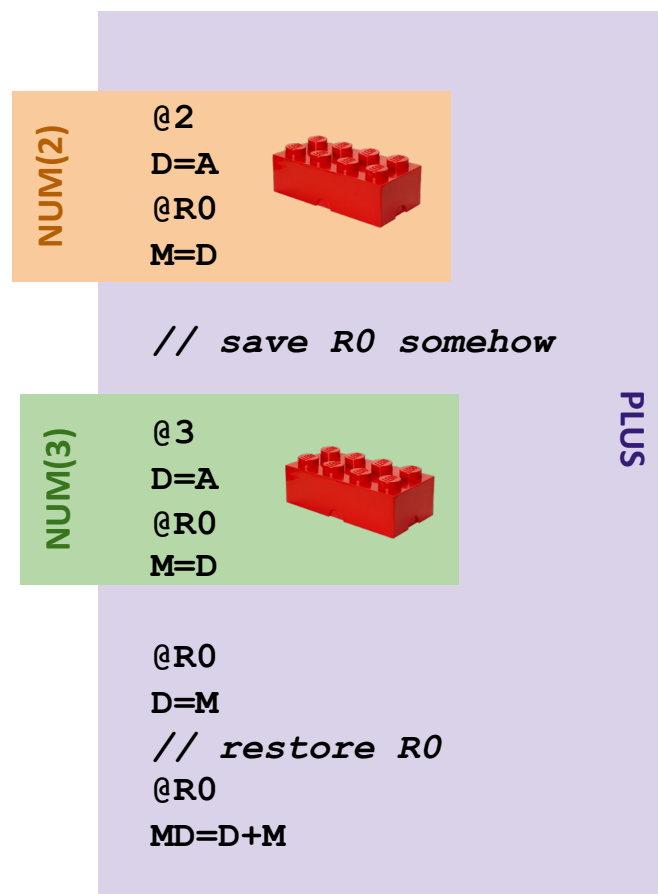
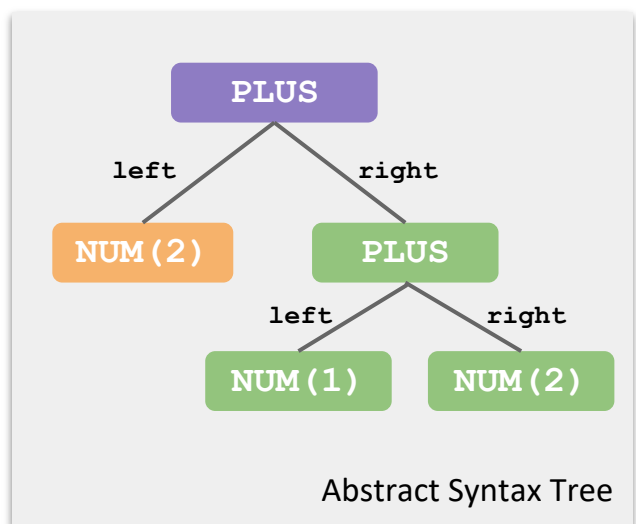


Computer
(actually, quite
clever!)



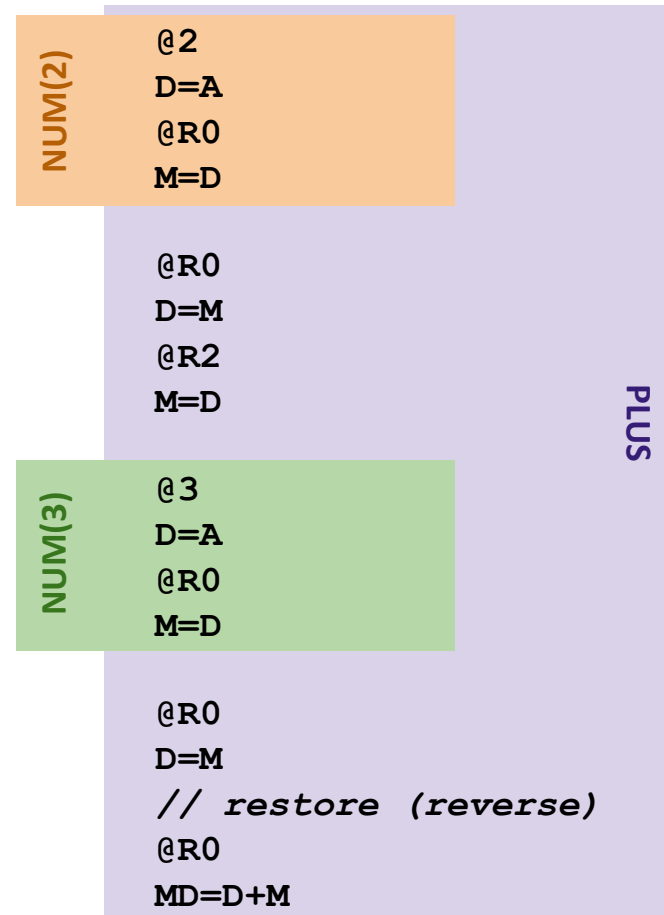
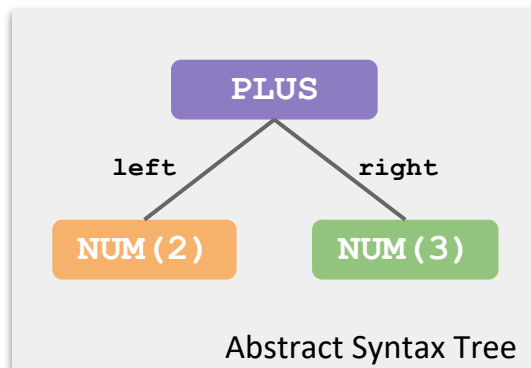
Code Generation: Example

- ❖ Why? Modularity: We can fit any expression in that slot, as long as **its result ends up in R0!**
 - Even another **PLUS**



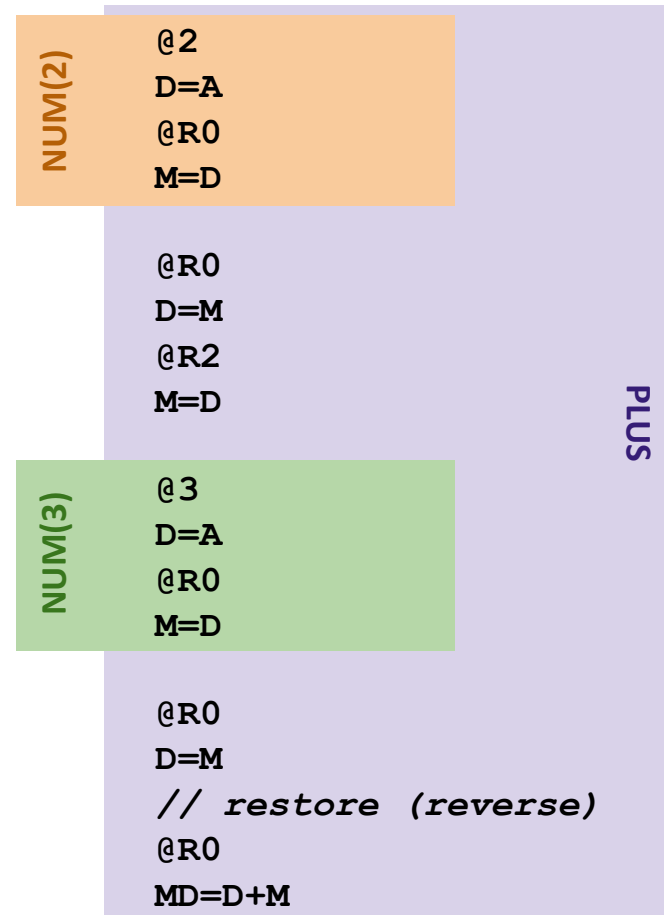
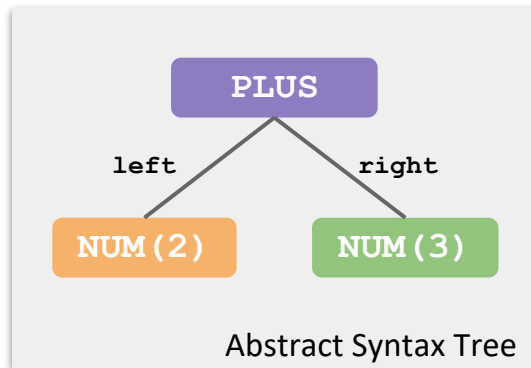
Code Generation: Example

- ❖ Now, we need to save R0 somehow
 - What if we save it in a temporary register? Let's pick R2



Code Generation: Example

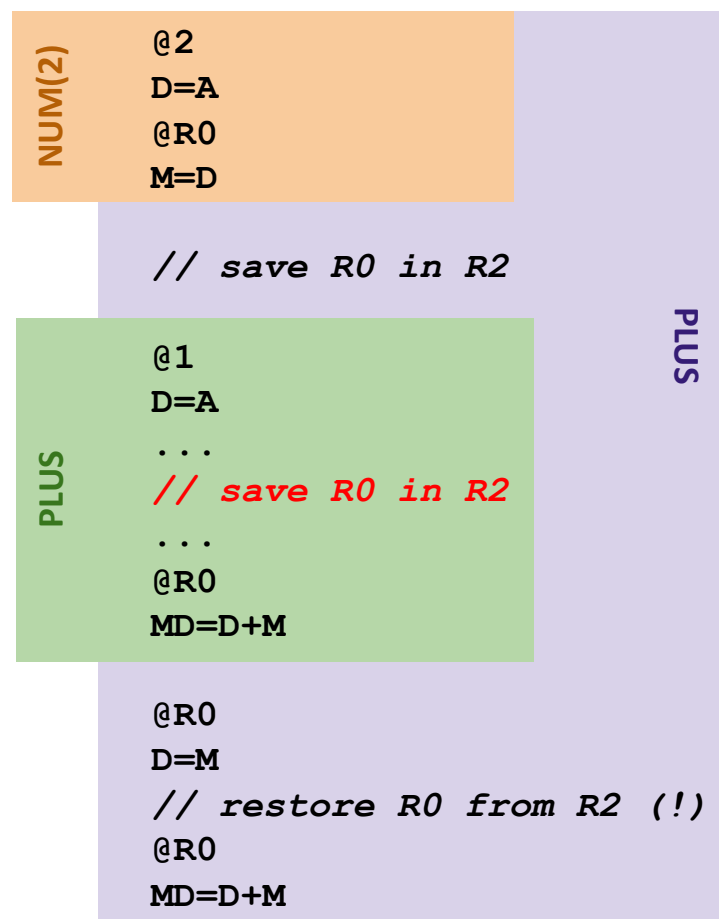
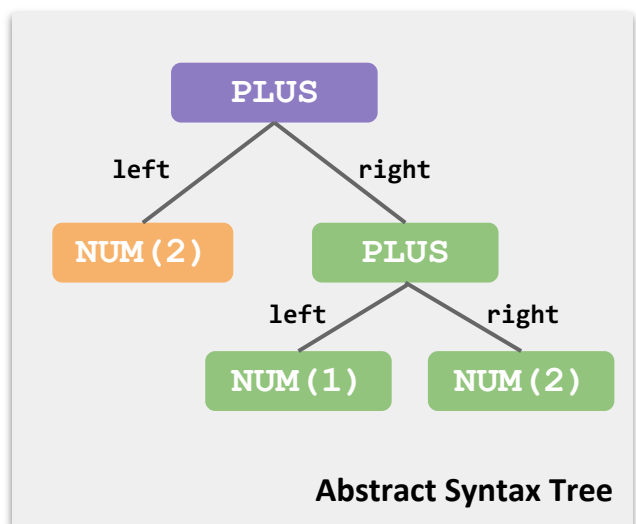
- ❖ Now, we need to save R0 somehow
 - What if we save it in a temporary register? Let's pick R2



Why won't this always work?

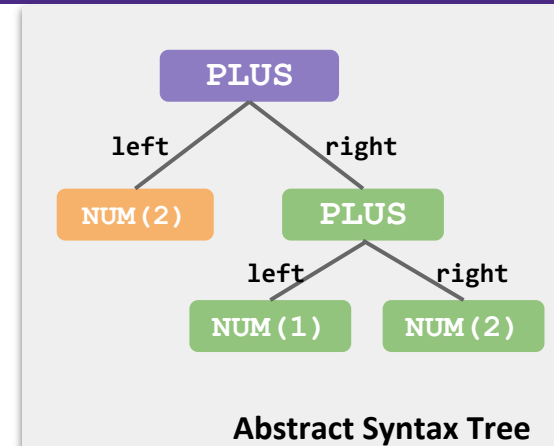
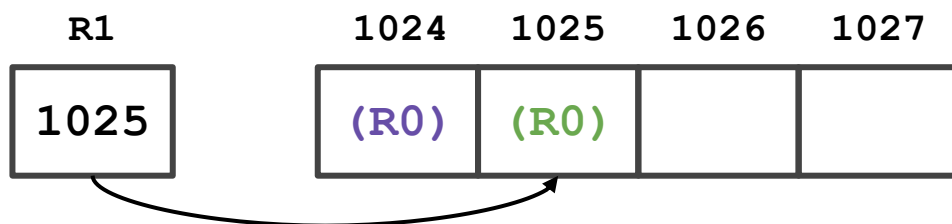
Code Generation: Example

- ❖ It's those pesky nested expressions! The **outer PLUS** saves a value in R2, but the **inner PLUS** overwrites that value during its computation



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



```

NUM(2)
@2
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
    
```

Code Generation: Example

❖ What about variables?

```
var int arr[5];  
var int bar, star;  
  
let bar = star;
```

Jack

```
@261  
D=M  
@262  
M=D
```

Hack Assembly

arr	256
bar	261
star	262
screen	16384

- ❖ Just like Assembler: Generate symbol table with mapping from variable names to spots in memory
 - Arrays get more (contiguous) spots
 - **screen** and **keyboard** are built-in array variables, allowing I/O

Code Generation: Takeaways

- ❖ Code Generation task: Writing several small snippets of Hack assembly
 - But need to be very generalizable
 - Whenever a PLUS expression is encountered, should generate almost the same code
- ❖ Conventions make the task much easier
 - For example, after any expression code runs, result should always be stored in R0
 - Then parent code can depend on it

Lecture Outline

- ❖ Debugging Metacognitive Skill
 - Debugging Process and Strategies, The Scientific Method
- ❖ Roadmap of Compilers
 - Scanner, Parser, Type Checker, Optimizer, Code Generation
- ❖ Compilers: Code Generation
 - Generating Target Code from AST
- ❖ **Two-Tier Compilation**
 - **Intermediate Programs and The Java Virtual Machine (JVM)**

Software Overview

Compiler
Project 8

High-Level Language

- Java
- Python
- C/C++
- Jack

Compiler

Intermediate Language(s)

- Java Byte Code
- Jack VM Code

Compiler (VM Translator)

Assembly Language

- x86, x86-64
- ARM
- RISC-V
- HACK

Operating System

- Windows
- Mac
- Unix/Linux
- Android
- Hack OS

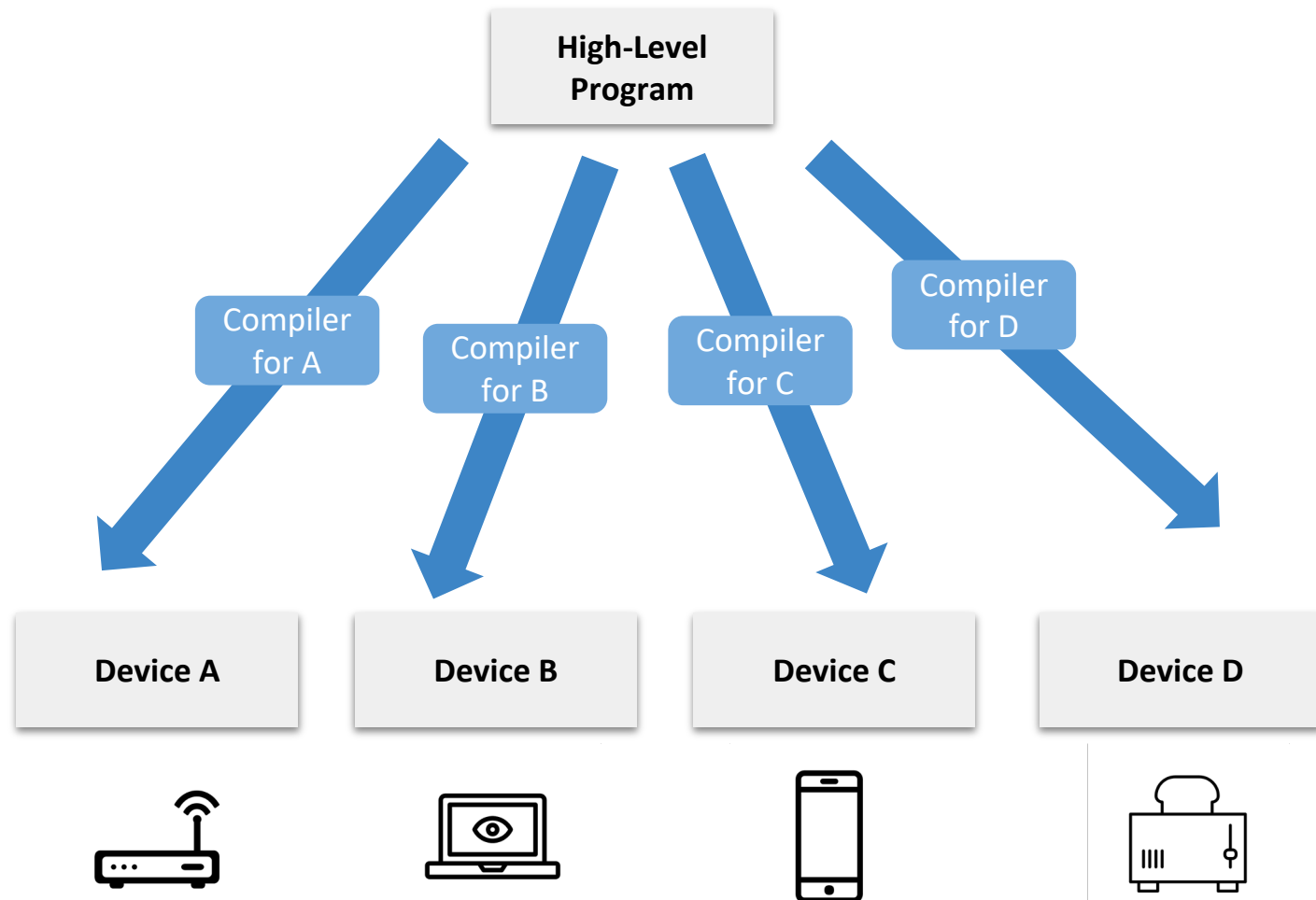
Assembler

Machine Code

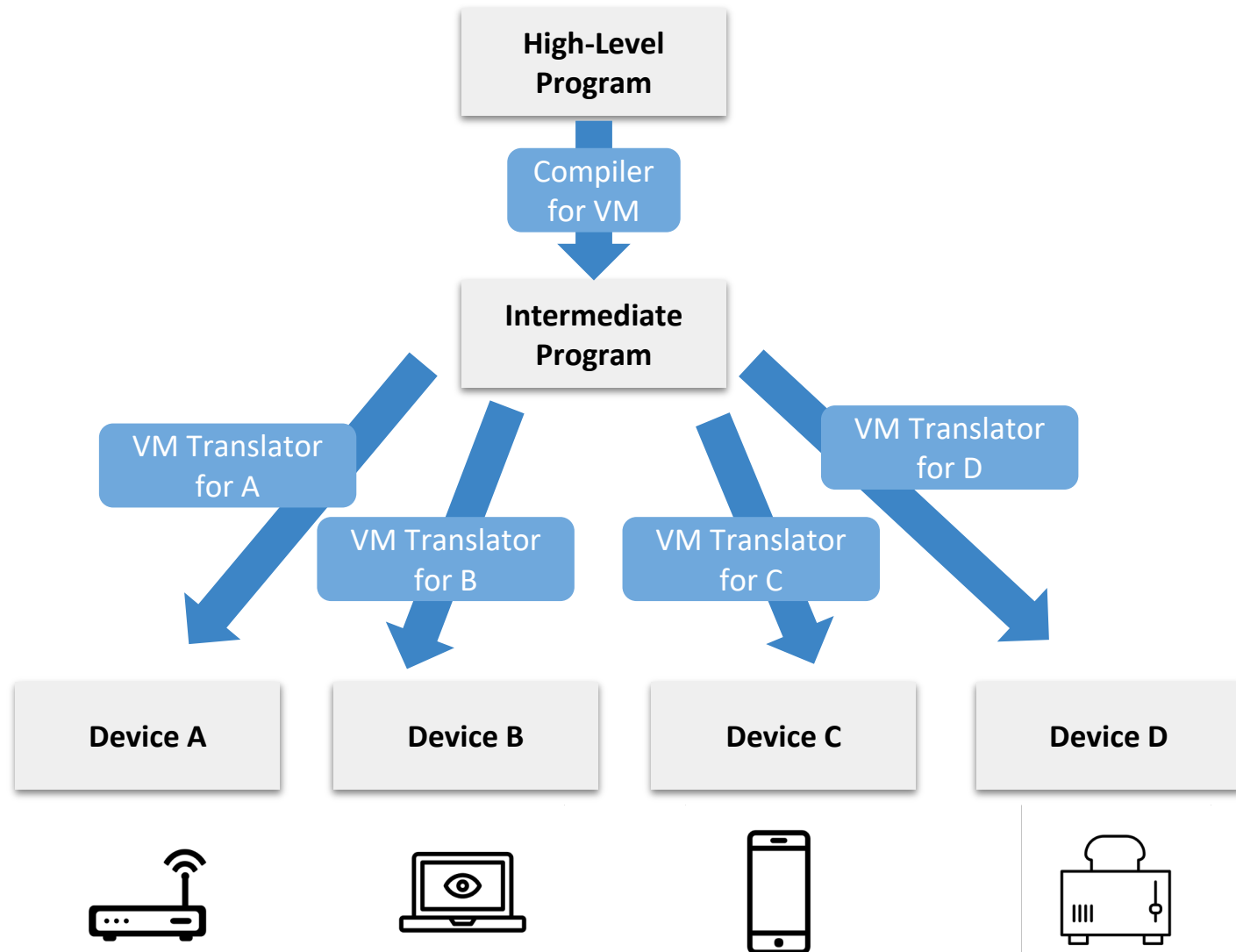
SOFTWARE

KEY: "Real-World" Examples
Our Computer

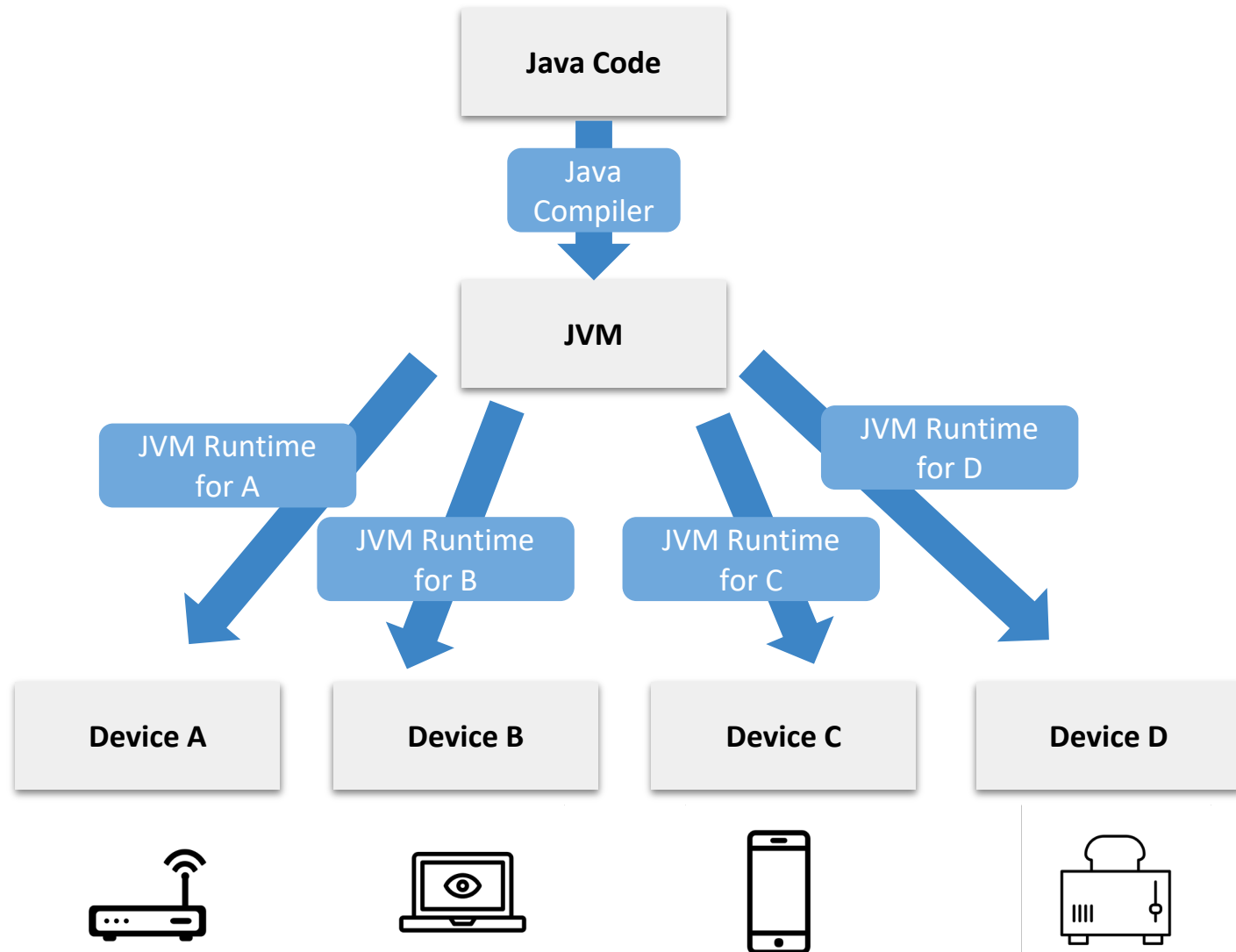
Compiling Code: Single Tier



Compiling Code: Two Tier



The Java Virtual Machine (JVM)



Lecture 15 Wrap-up

❖ Project Reminders

- **Project 7, Part I (Midterm Corrections) due this Thursday (5/19)**
- Project 7, Part II (Professor Meeting Report) due next Thursday (5/26)

❖ Thursday's Lecture Pre-reading: Stress & Wellness Podcast

- See calendar for link to podcast and discussion prompts