CSE 390 B Spring 2021

Compiler Part II, Social Computing Reflection II

Compiler Code Analysis and Generation, Social Computing Reflection II, TA Feedback

Agenda

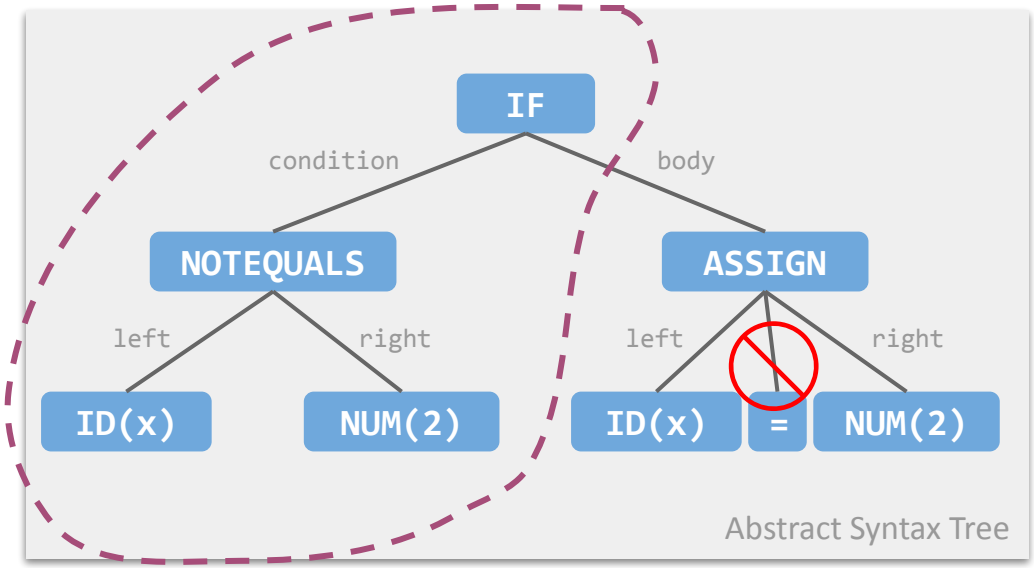
- Reading Review and Q&A
- Compiler Code Generation
- Project 7 Tools Demo/Practice
- Social Computing Reflection II Discussion
- TA Feedback Form
- Reminders

Agenda

- **Reading Review and Q&A**
- Compiler Code Generation
- Project 7 Tools Demo/Practice
- Social Computing Reflection II Discussion
- TA Feedback Form
- Reminders

Review: Abstract Syntax Trees

- Composed of nodes that capture the **structure** of input program
  - Can be recursive! (And almost always is)
- *Important distinction:* cares about big-picture syntax (e.g. entire `if` statement) rather than nitty-gritty syntax (e.g. semicolons, parentheses, even word "if" used to write that `if` statement)



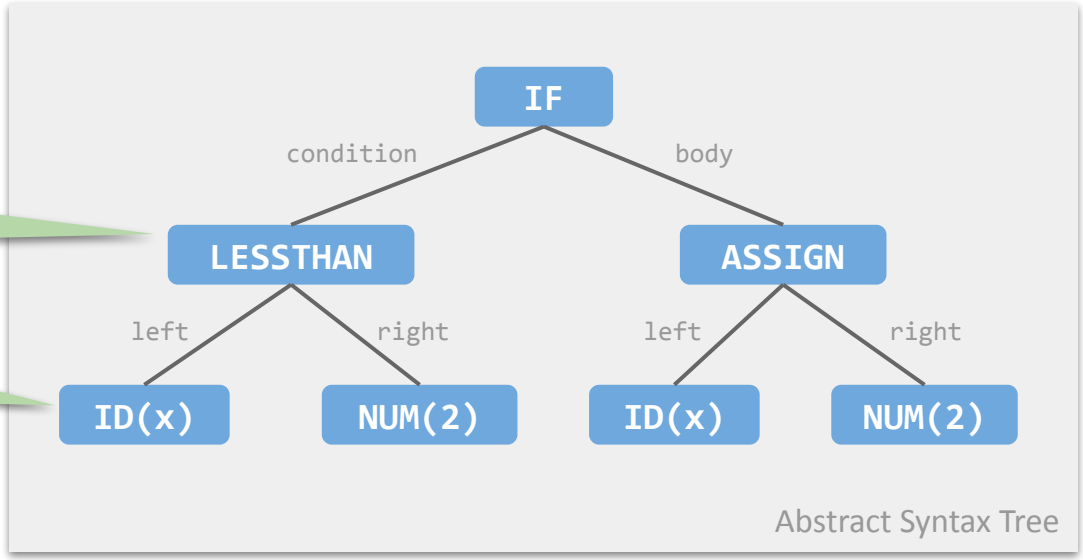

```
class If {
  Expression condition; //stores a NotEquals
  List<Statement> body;
}

class NotEquals extends Expression {
  Expression left; //stores a VarAccess
  Expression right; //stores a NumLiteral
}

class VarAccess extends Expression {
  Identifier name; //stores x
}

class NumberLiteral extends Expression {
  int value; //stores 2
}
```

Type Checking (Semantic Analysis)

- Given the abstract syntax tree, run checks over it to ensure that it fits within constraints of the language
  - Do the types match up?
- Collect additional info for code generation, such as number/type of arguments in each function

Does this expression evaluate to a boolean?

Is the variable "x" defined at this point?



Optimization

- Code improvement: change correct code into *semantically equivalent* but "better" code
  - Example: if something is computed every iteration of a while loop, the compiler could yank that computation out and compute it just once before entering the loop
    - Here, "better" means faster
  - But requires caution: what if the value changes on each iteration of the loop?
    - "Semantically equivalent" means user sees same outcome

AST Recursive Pattern

- Vast number of possible AST configurations
- Recursion makes doing tasks with ASTs much easier!
- Rather than think of all the possible children configurations, can follow a simple pattern:
  - Recursively have children nodes do their work for the task
  - Do the work for the current node, often synthesizing/making use of results from child nodes

AST Recursive Pattern: Type Checking

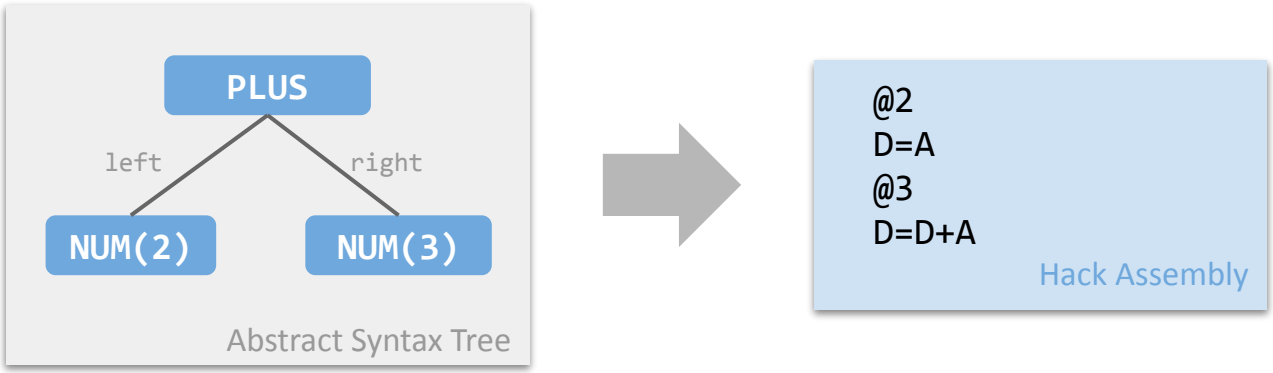
```
class Add {
    Exp left;
    Exp right;

    typeCheck() {
        // Recursively type check children and get their type
        leftType = left.typeCheck()
        rightType = right.typeCheck()

        // Check for valid type combinations for add
        if (leftType == int && rightType == int) {
            return int
        } else if (leftType == int && rightType == double) {
            return double
        } else if... {
            // continue cases for valid types/returning resulting type
        }

        // if we reach this case then we have a type pair that isn't valid for add
        throw new TypeError("Incompatible types for Add")
    }
}
```

Reading Q&A

Agenda

- Reading Review and Q&A
- **Compiler Code Generation**
- Project 7 Tools Demo/Practice
- Social Computing Reflection II Discussion
- TA Feedback Form
- Reminders

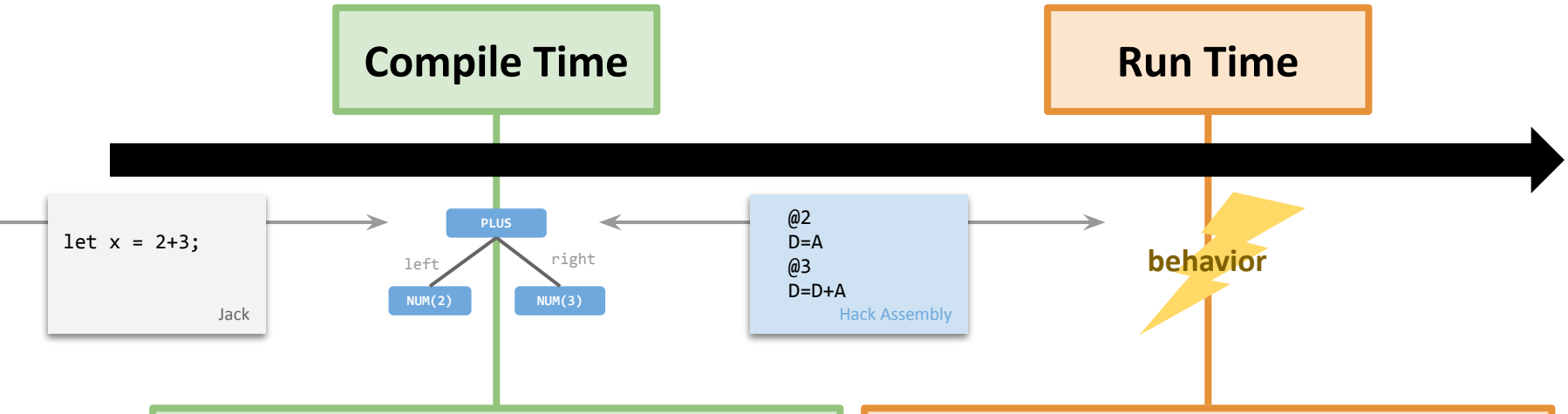
Code Generation: The Task



- Convert the AST into **target language code** that produces the same result

- 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

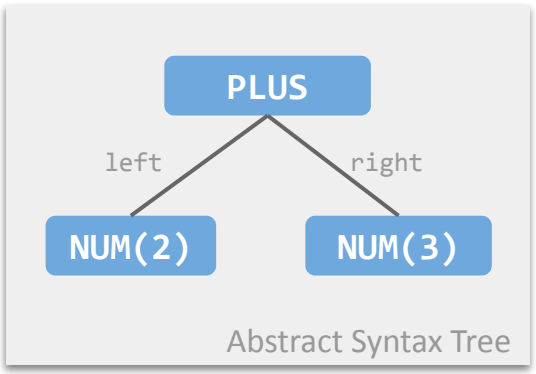


- Compiler (a Java program) is running
  - Generates Hack instructions that will be run later
- Know types of variables, but **DO NOT KNOW value of variables, which code path is taken**

- Output (a Hack program) is running on the Hack computer
- Know value of variables, which code path is taken

Code Generation: Example

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



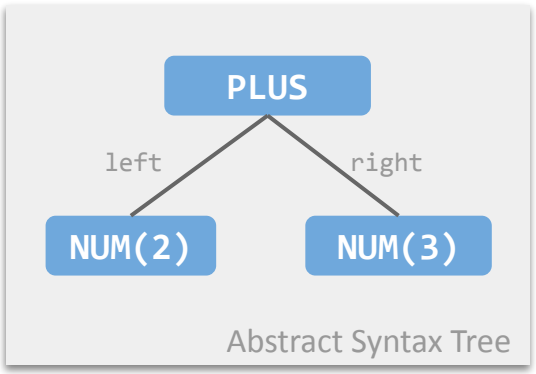
Human (practically a genius)

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

Hack Assembly

Code Generation: Example

- Here's how the computer sitting in front of you might do it, instead:



Human (practically a genius)

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

Hack Assembly

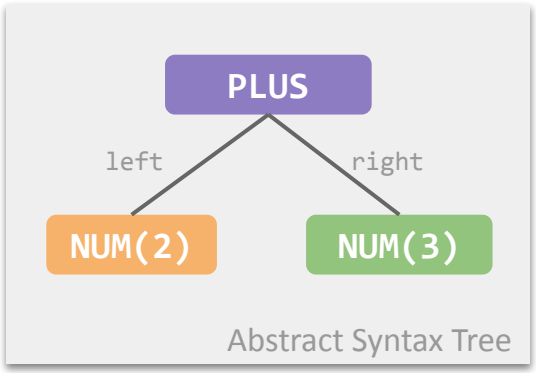
Computer (trying its best)

```
@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 pretty clever?!)

PLUS

NUM(2)
```
@2
D=A
@R0
M=D
```

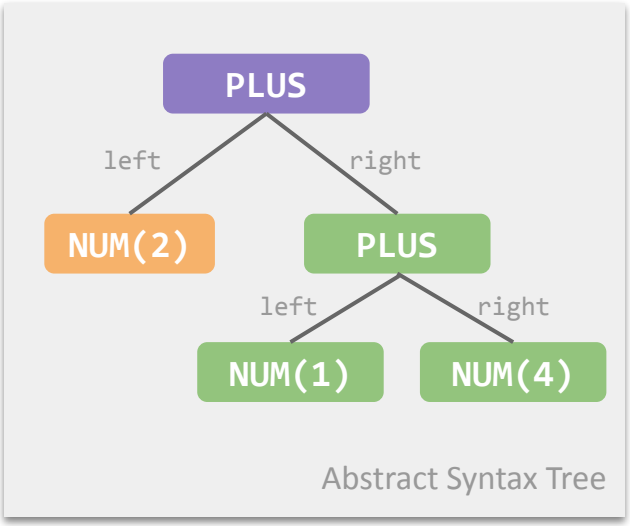
```
// save R0 somehow
```

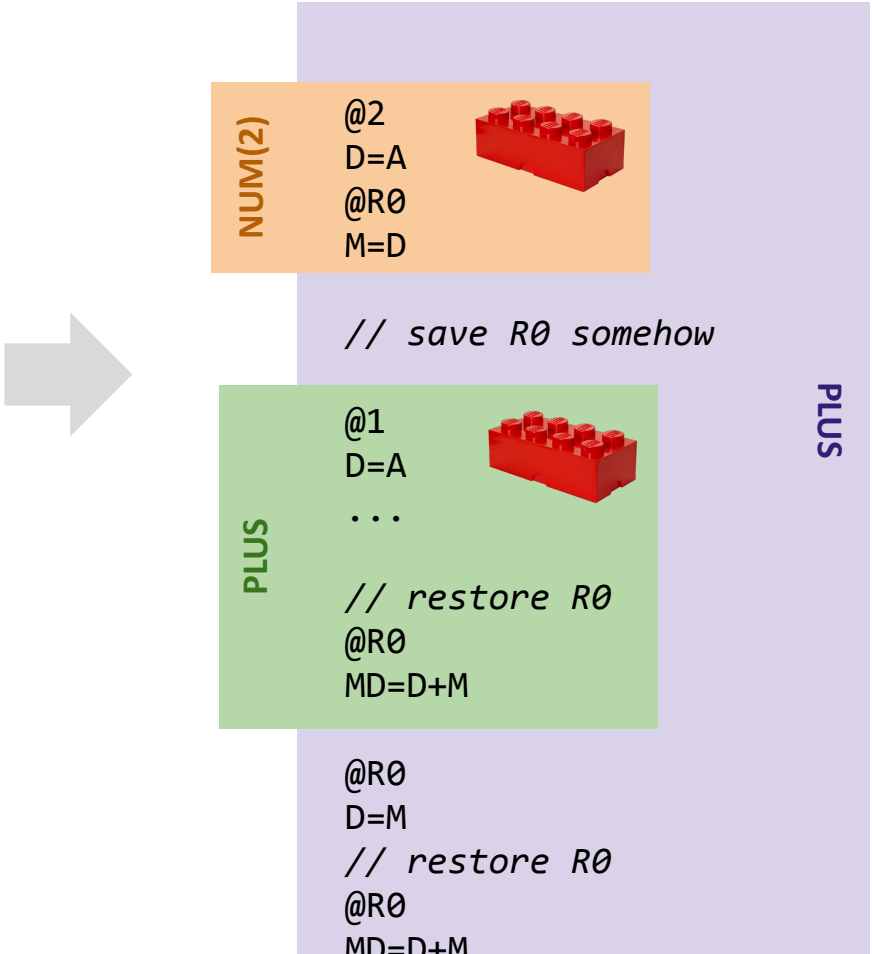
NUM(3)
```
@3
D=A
@R0
M=D
```

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

Code Generation: Example

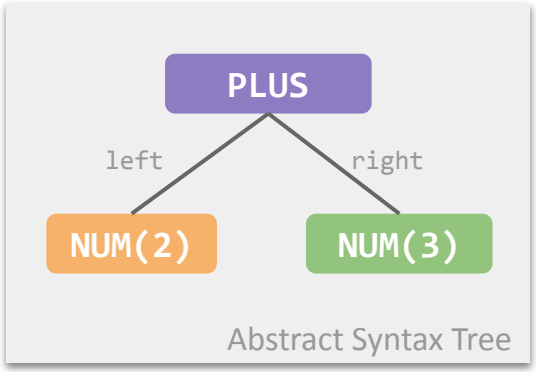
- Why? Modularity: we can fit *any* expression in that slot, as long as **its result ends up in R0!**
  - Even another PLUS!





Code Generation: Example

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

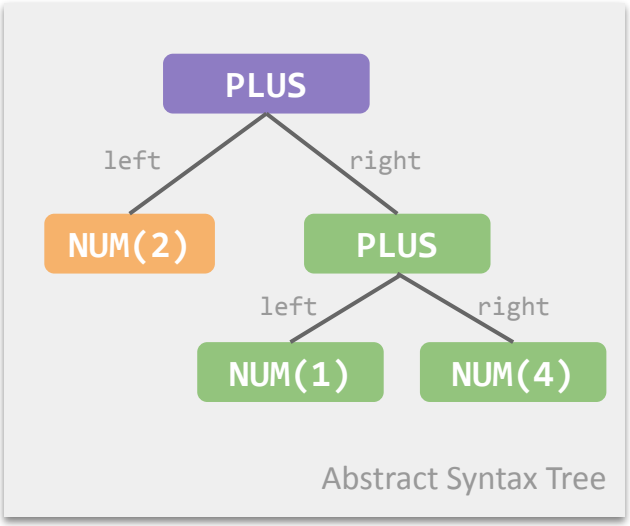


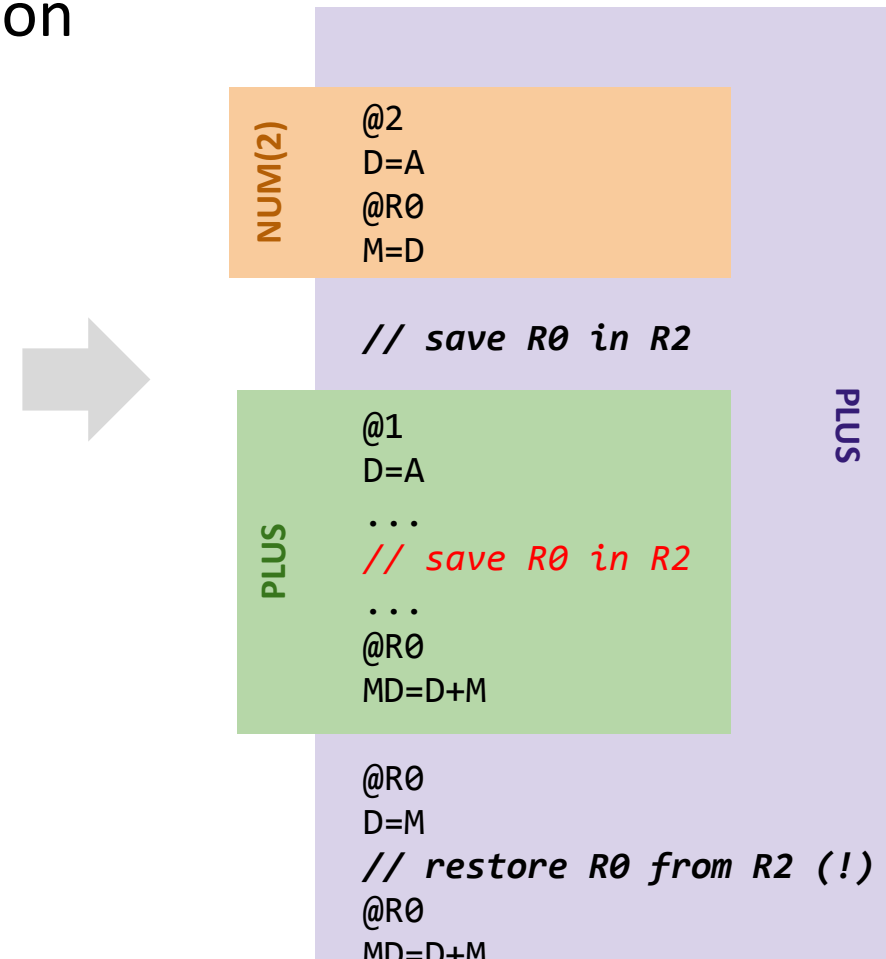

```
NUM(2)
@2
D=A
@R0
M=D

@R0
D=M
@R2
M=D

NUM(3)
@3
D=A
@R0
M=D

PLUS
@R0
D=M
// restore (reverse)
@R0
MD=D+M
```

- Why won't this always work?

Code Generation: Example

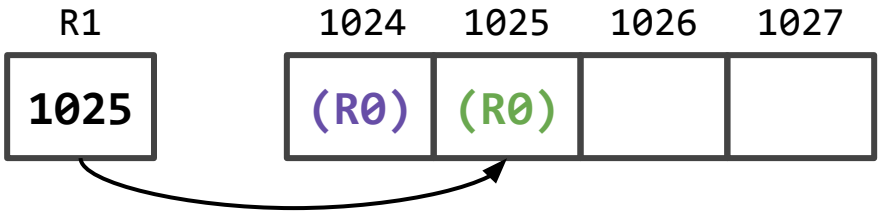
- It's those pesky nested expressions! The outer PLUS saves a value in R2, but the inner PLUS overwrites that value during *its* computation





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 will be our "stack pointer": always stores address of the 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
```

PLUS

```
@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
  - E.g. after any expression code runs, **result should always be stored in R0**. Then parent code can depend on it!
- Beware of **clobbering** data
  - Use our little stack as necessary!
  - `Plus.java` contains an example of pushing/popping
- More Project 7 specifics later

Agenda

- Reading Review and Q&A
- Compiler Code Generation
- **Project 7 Tools Demo/Practice**
- Social Computing Reflection II Discussion
- TA Feedback Form
- Reminders

Project 7 Overview

- Part I: Buggy Compiler!
  - Code Generation Implementation/Debugging
- Part II: Social Reflection III

Project 7 will be due **Tuesday (6/1)**! Please note the shift in due date!

(You're not expected to start project 7 until after midterm corrections are turned in this Thursday)

Project 7 Part I: Buggy Compiler

- You will be given starter code for a compiler that reads a micro version of Jack and spits out Hack!
  - The **Scanner** & **Parser** work
    - Task A: Read through to understand what's going on. There are heavy comments to help you out!
  - The **Code Generation** is buggy and half-finished
    - Task B: Find the bugs by practicing deliberate debugging strategies. Remember you can step through generated Hack code using the CPUEmulator tool.
    - Task C: Finish the compiler! These slides will be a helpful reference.

Project 7: Micro Jack

- Stripped-down version of Jack language
  - More manageable but enough features to be interesting!
- Has:
  - Types: Int and Int[], Variable Assignment, If, While, +, -, ==, !=
- Doesn't Have:
  - Functions/Function Calls, Classes/Objects, Strings, For, Array Bounds Checking, etc.

Any number of variable declarations

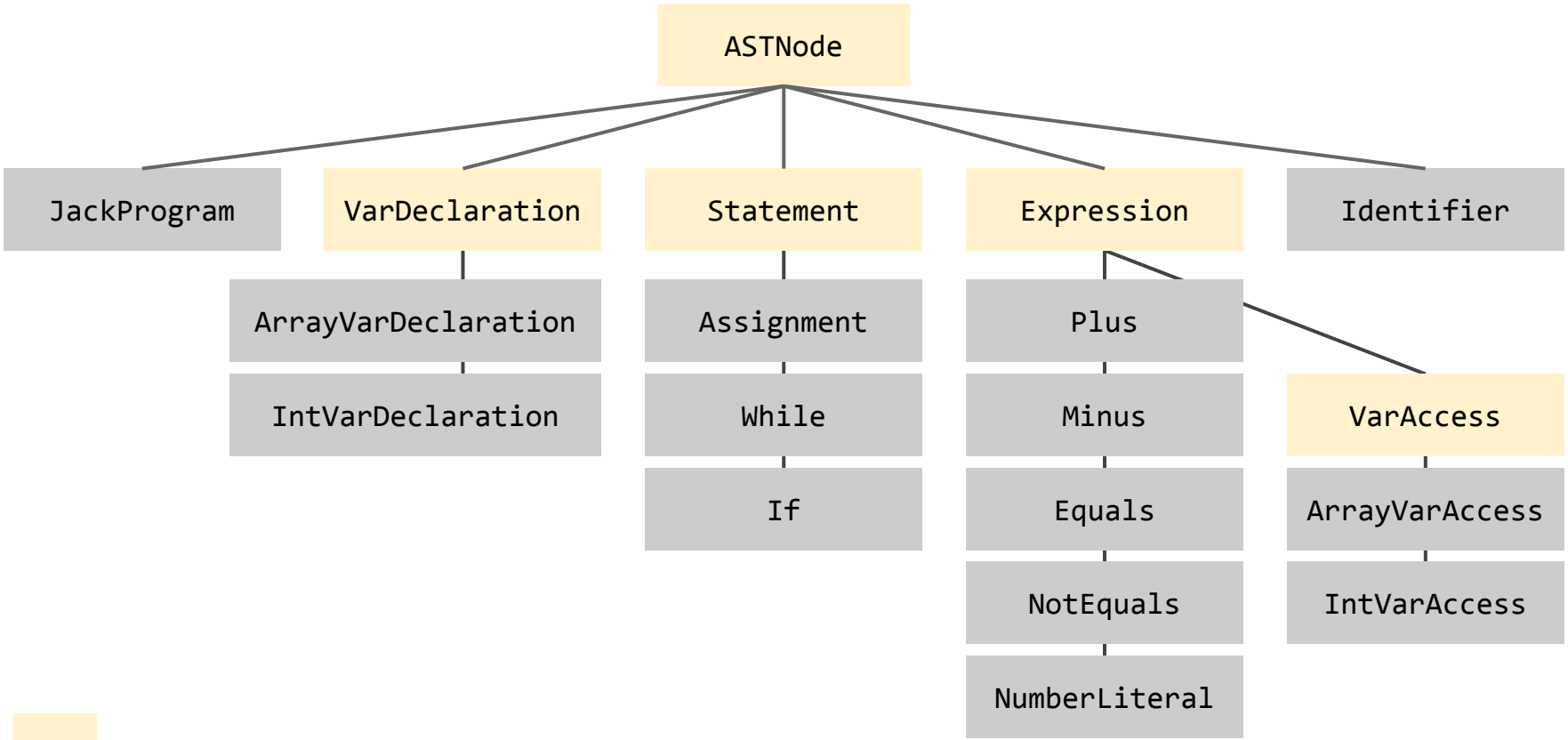
Basic.jack

```
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 7: The AST Nodes

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



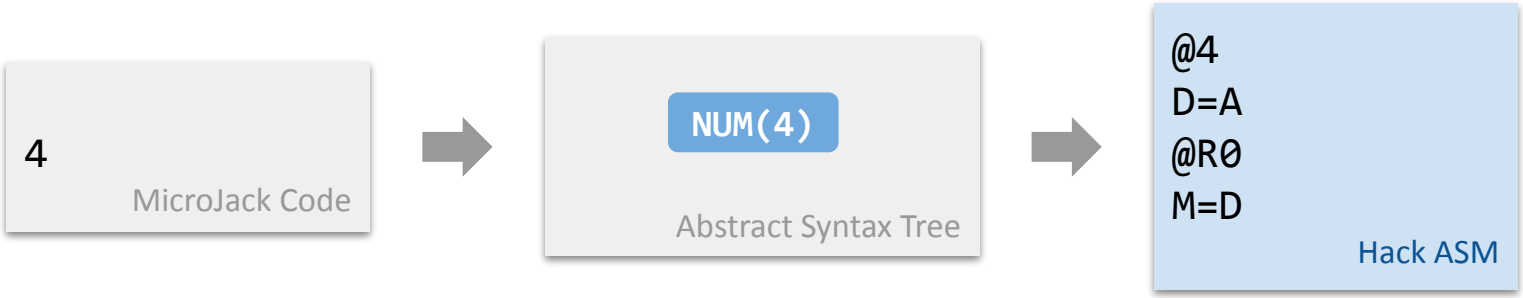
Project 7: Generating Code

- 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");

  ...

}
```

Example: Number Literal (Step 1)

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



Example: Number Literal (Step 1)

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

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

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

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

4

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

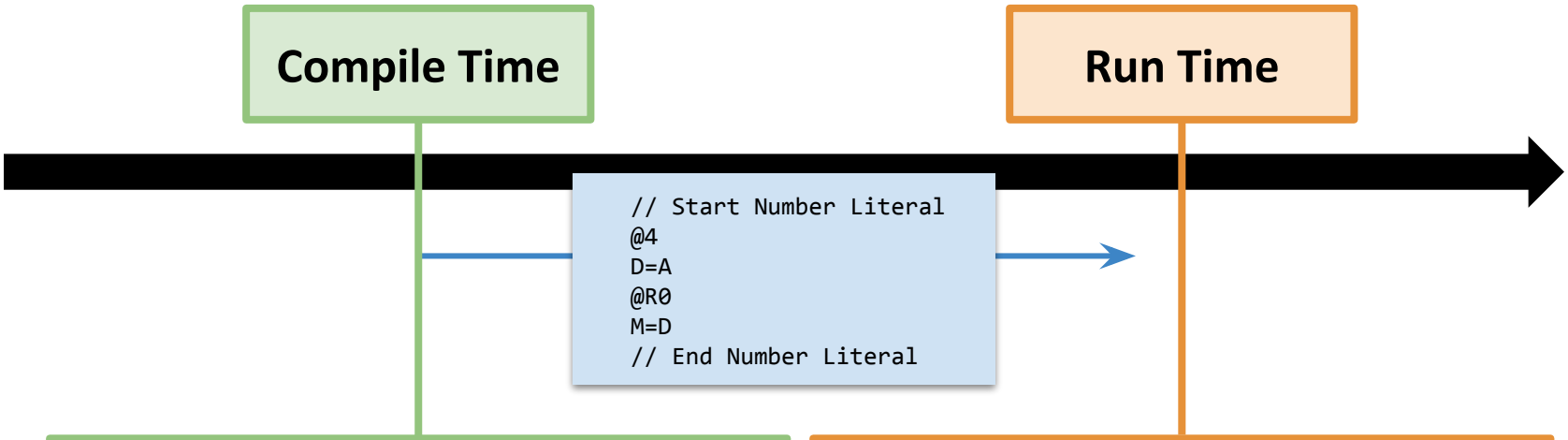
Example: Number Literal (Step 1)

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

    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);
    }
}
```

4

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

Example: Number Literal (Step 1)



- Java (the compiler) is running
- 4 is stored as value field inside an NumberLiteral ASTNode
- When executed, **prints code that stores it another way**!

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

    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);
    }
}
```

- Hack ASM (the output) is running
- 4 is stored as constant inside an assembly instruction
- When executed, loads 4 from instruction into A register

Project 7 Demo

Brief project 7 tools demo

Project 7 Tools Practice

- Practice using the project 7 tools! Try doing the following:
  - Run **`git pull`** to grab the project 7 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 program

    `java compiler/Compiler compile ../test/OnlyVars.jack`
  - Load/run OnlyVars.tst in the CPUEmulator

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

Agenda

- Reading Review and Q&A
- Compiler Code Generation
- Project 7 Tools Demo/Practice
- **Social Computing Reflection II Discussion**
- TA Feedback Form
- Reminders

Social Computing Reflection II Discussion

- Share the article you found for your second social reflection!

- Things to include:
  - Give a summary of the article
  - Share how it changed/influenced your thinking
  - Share your follow up questions
  - Feel free to have discussions beyond these bullet points too!

Agenda

- Reading Review and Q&A
- Compiler Code Generation
- Project 7 Tools Demo/Practice
- Social Computing Reflection II Discussion
- **TA Feedback Form**
- Reminders

1:1 Meetings & TA Feedback

Your TA's want to hear from you!

Please take a few minutes to reflect on how your 1:1 TA meetings have been going and complete the anonymous feedback form in the zoom chat.

Agenda

- Reading Review and Q&A
- Compiler Code Generation
- Project 7 Tools Demo/Practice
- Social Computing Reflection II Discussion
- TA Feedback Form
- **Reminders**

Reminders

- **Midterm Redo** - due this Thursday
  - Open-note, open-tool. Midterm questions are on Ed Board
  - Midterm grade will be the average of your initial midterm grade and your redo score; your midterm grade can *only improve*
- **Professor Meeting Report** - due in 1.5 weeks
  - Identify and schedule your professor meeting ASAP.
- Thursday's Reading: A Podcast!!!
  - ~25min but hopefully a nice break from readings!
  - See calendar for link/discussion prompts