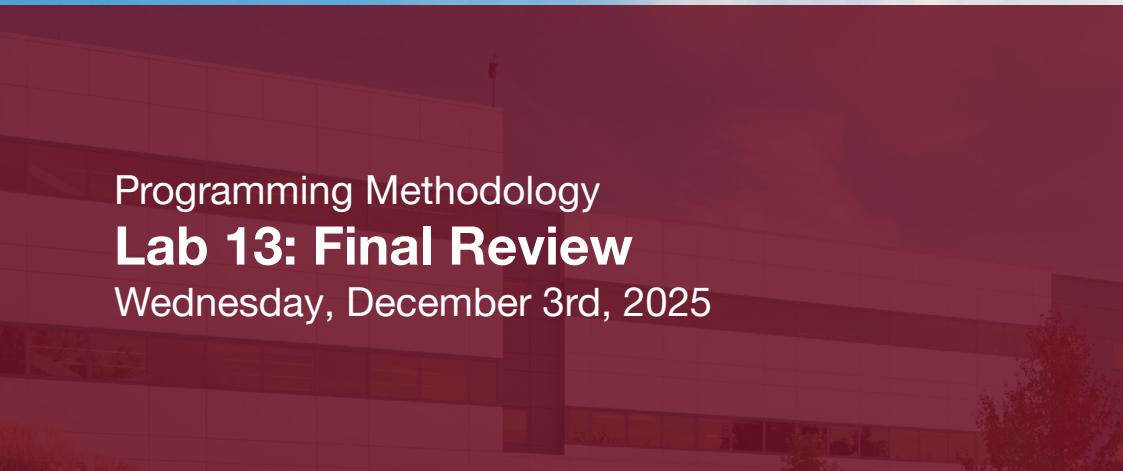




Manning College of Information
& Computer Sciences

Programming Methodology
Lab 13: Final Review
Wednesday, December 3rd, 2025



Weekly Lab Agenda

- Go over reminders/goals
- Review past material
- Work in groups of 2-3 to solve a few exercises
 - Lab leaders will assign new groups this week
- Discussion leaders will walk around and answer questions
- Solutions to exercises will be reviewed as a class
- Attendance taken at the end

Reminders

- Great job this semester everyone, you should be proud of your hard work!
 - Best of luck in these final couple of weeks!
- Final exam Logistics:
 - Tuesday, December 16th, 3:30pm - 5:30pm
- Please fill out the SRTI course survey, this really helps make the class better! You can find a link to this on Canvas.
- Homework 7 is due this Sunday (12/7) at 11:59pm
- Two more graded requirements for homework 6:
 - Homework 6 Team Evaluations are due today
 - Homework 6 Self Evaluations will be released by tomorrow and due later this week

Today's Goals

- Review select topics for the final exam
 - Program correctness
 - Interpreters
 - Please review lectures, labs, and past exams for more practice on other topics covered this semester
- Fill out feedback form for 220 labs from this semester!
 - Please fill this out! We would love to hear from you as we aim to improve these labs in future semesters

What's an Invariant?

- As its name implies, an invariant is something that *does not change*
 - Here: a *specific type of assertion* (not all assertions are invariants)
- We will work with **loop invariants**: they hold **before** each loop iteration (before checking the condition)
 - A useful invariant relates variables that *change* within the loop
 - The invariant will not hold at *every* point in the loop but will be *restored* (holds again) at the end

Exercise 1: Program Correctness

Write a function `maxInInterval(A, start, stop)`, where **A** is an array of integers, **start** is an integer ≥ 0 , and **stop** is an integer $\geq \text{start}$. Both **start** and **stop** are assumed to be valid indices for array **A**.

`maxInInterval(A, start, stop)` returns the largest number found in array **A** between positions **start** and **stop**, both inclusive.

While writing this function, state your invariants. Show how your invariants change throughout your function. Try to use a while loop in your solution and show that the loop invariant is reestablished before each new iteration.

Exercise 1: Solution

```
function maxInInterval(A: number[], start: number, stop: number): number {  
    let current_max = ?  
    // ...  
}
```

We will begin by considering how we want to generally approach this problem. It makes sense for us to iterate through the given array, from **start** to **stop** (inclusive), keeping track of the maximum we have seen so far. We will therefore declare a variable `current_max` to track this.

What should our invariant be? What do we want `current_max` to be as we iterate through `A` from **start** to **stop**?

Exercise 1: Solution

```
function maxInInterval(A: number[], start: number, stop: number): number {  
    let current_max = A[start];  
    // ...  
}
```

Informally, our invariant will be: `current_max` is the largest element we have seen so far.

Thus, as we will be iterating from **start** to **stop**, we will initially define `current_max` to be our initial value `A[start]`.

Exercise 1: Solution

```
function maxInInterval(A: number[], start: number, stop: number): number {  
    let current_max = A[start];  
    let next = start + 1;  
    // ...  
}
```

To make our invariant more formal, we will also define a variable **next** to be the next index in **A** that we would like to examine.

Our invariant will therefore be: **current_max** is the largest element in **A** between **start** and **next - 1** (inclusive).

We see this clearly holds when we initially set **next** to be **start + 1**, as **current_max = A[start]** is the maximum value between **start** and **start**.

Exercise 1: Solution

```
function maxInInterval(A: number[], start: number, stop: number): number {  
    let current_max = A[start];  
    let next = start + 1;  
    while (next <= stop) {  
        // ...  
    }  
    // ...  
}
```

We will now iterate through the array A until we have reached index **stop**. We use `<=` in order to include the element at index **stop**.

Our invariant going into this loop is still: **current_max is the largest element in A between start and next - 1 (inclusive)**.

However, we now want to update our `current_max`, based on `A[next]`.

Exercise 1: Solution

```
function maxInInterval(A, start, stop) {  
    let current_max = A[start];  
    let next = start + 1;  
    while (next <= stop) {  
        if (A[next] > current_max) {  
            // ...  
        } else {  
            // ...  
        }  
    }  
    // invariant: current_max is the largest  
    // element in A between start and next - 1  
    // (inclusive)  
  
    // removed type annotations to save  
    // space, don't forget them on the exam :)
```

We now consider two cases: either $A[\text{next}]$ is larger than our `current_max`, or it is not.

Within the if block, we know that $A[\text{next}]$ is larger than `current_max`. Using our invariant, this means $A[\text{next}]$ is larger than all the elements in A between start and $\text{next} - 1$ (inclusive).

Consequently, this makes $A[\text{next}]$ the largest element in A between start and next (inclusive).

Exercise 1: Solution

```
function maxInInterval(A, start, stop) {  
    let current_max = A[start];  
    let next = start + 1;  
    while (next <= stop) {  
        if (A[next] > current_max) {  
            current_max = A[next];  
        } else {  
            // ...  
        }  
        // ...  
    }  
    // ...  
  
    // invariant: current_max is the largest  
    // element in A between start and next - 1  
    // (inclusive)  
}
```

Since we found a larger value than `current_max`, we will update `current_max` accordingly.

After this statement, our invariant is no longer true. `current_max` is no longer an element of `A` between `start` and `next - 1` (inclusive), thus our invariant no longer holds by the end of this if block.

Exercise 1: Solution

```
function maxInInterval(A, start, stop) {  
    let current_max = A[start];  
    let next = start + 1;  
    while (next <= stop) {  
        if (A[next] > current_max) {  
            current_max = A[next];  
        } else {  
            // do nothing!  
        }  
        // ...  
    }  
    // ...  
  
    // invariant: current_max is the largest  
    // element in A between start and next - 1  
    // (inclusive)  
}
```

Now we can consider the else block. Within this else block, we know that the condition $A[next] > current_max$ is false.

Thus, it must be that $current_max \geq A[next]$.

Our invariant still holds, and we additionally notice that $current_max$ is also the largest element in A between $start$ and $next$ (inclusive).

We therefore do not actually need to do anything in this else block!

Exercise 1: Solution

```
function maxInInterval(A, start, stop) {  
    let current_max = A[start];  
    let next = start + 1;  
    while (next <= stop) {  
        if (A[next] > current_max) {  
            current_max = A[next];  
        } else {  
            // do nothing!  
        }  
        // what should we do here?  
    }  
    // ...  
  
    // invariant: current_max is the largest  
    // element in A between start and next - 1  
    // (inclusive)
```

After the entire if-else statement,
what do we know?

As we reasoned, both branches
ultimately result in `current_max`
becoming the largest element in `A`
between `start` and `next` (inclusive).

We now want to use this to
reestablish our invariant.

Exercise 1: Solution

```
function maxInInterval(A, start, stop) {  
    let current_max = A[start];  
    let next = start + 1;  
    while (next <= stop) {  
        if (A[next] > current_max) {  
            current_max = A[next];  
        } else {  
            // do nothing!  
        }  
        next = next + 1;  
    }  
    return current_max;  
}
```

```
// invariant: current_max is the largest  
// element in A between start and next - 1  
(inclusive)
```

After incrementing `next`, `current_max` is now the largest element in `A` between `start` and `next - 1` (inclusive).

We have therefore reestablished our invariant at the end of the while loop.

Lastly, we return `current_max` after the while loop terminates.

Exercise 1: Solution

```
function maxInInterval(A, start, stop) {  
    let current_max = A[start];  
    let next = start + 1;  
    while (next <= stop) {  
        if (A[next] > current_max) {  
            current_max = A[next];  
        }  
        next = next + 1;  
    }  
    return current_max;  
}  
  
// invariant: current_max is the largest  
// element in A between start and next - 1  
// (inclusive)
```

Note that we did not reason with invariants why the while loop must terminate; this will be explored more in lecture tomorrow!

We can also omit the else statement entirely, as it does nothing. It can be nice to include it though (or at least imagine it) when you are analyzing invariants in a function.

Exercise 1: Solution

```
function maxInInterval(A: number[], start: number, stop: number): number {  
    // invariant: current_max is the largest element in A between start and next - 1 (inclusive)  
    let current_max = A[start];  
    let next = start + 1;  
    // current_max is the largest element in A between start and next - 1 = start (inclusive)  
    while (next <= stop) {  
        if (A[next] > current_max) {  
            // A[next] is larger than current_max, so A[next] is the largest element in A between  
start and next (inclusive)  
            current_max = A[next];  
            // current_max is the largest element in A between start and next (inclusive)  
        } else {  
            // A[next] <= current_max, so current_max is the largest element in A between start  
and next (inclusive)  
        }  
        // current_max is the largest element in A between start and next (inclusive)  
        next = next + 1;  
        // current_max is the largest element in A between start and next - 1 (inclusive)  
    }  
    return current_max;    This is an example of how you might answer this question, implementing  
the function and writing invariants as comments.  
}
```

Note: *slightly* harder than what you can expect
as an exam question!

UMassAmherst

Manning College of Information
& Computer Sciences

Exercise 2: Interpreters

You are given an expression AST containing only numbers and the binary operators "+" and "-". The given expression AST also contains exactly one variable node, corresponding to the variable x. The property `containsX` has also been added to each AST node, indicating if the variable x is within the AST whose root is that node.

This means we define the Expression type in this problem as follows:

```
type Expression = { kind: "number", value: number, containsX: false }
| { kind: "variable", name: "x", containsX: true }
| { kind: "operator", operator: "+" | "-", left: Expression,
  right: Expression, containsX: boolean };
```

Exercise 2: Interpreters

Write a function `solveForX(e: Expression, result: number): number` that returns the value that `x` must be in order for the given expression `e` to evaluate to `result`.

You may use the function `evalExpression(e: Expression)`, which will evaluate a given expression AST that does not contain any variables (though you should feel comfortable implementing this yourself too!).

```
type Expression = { kind: "number", value: number, containsX: false }
| { kind: "variable", name: "x", containsX: true }
| { kind: "operator", operator: "+" | "-", left: Expression,
  right: Expression, containsX: boolean };
```

Exercise 2: Solution

```
function evalExpression(e: Expression): number {  
    if (e.kind === "operator") {  
        const left = evalExpression(e.left);  
        const right = evalExpression(e.right);  
        return e.operator === "+" ? left + right : left - right;  
    } else if (e.kind === "number") {  
        return e.value;  
    } else {  
        throw new Error("AST contains variables.");  
    }  
}
```

You did not need to implement this, but you should definitely feel comfortable implementing a function like this.

Throwing an error is optional, as we define `evalExpression` only for Expressions with no variables. In general, you do not need to throw an error unless we explicitly ask you to.

Exercise 2: Solution

```
function solveForX(e: Expression, result: number): number {  
    if (e.kind === "variable") {  
        return result;  
    } else {  
        // ...  
    }  
    // ...  
}
```

We are told the given Expression contains the variable x. This means it is either the variable x or an operator.

If it is a variable, then we know it is the variable x, as we are told there are no other variables in the given Expression.

We therefore return **result**, as our expression is just x, so setting x = result allows the Expression **e** to evaluate to **result**.

Exercise 2: Solution

```
function solveForX(e: Expression, result: number): number {  
    if (e.kind === "variable") {  
        return result;  
    } else {  
        const leftHasX = e.left.containsX;  
        const sideWithX = leftHasX ? e.left : e.right;  
        const sideWithoutX = leftHasX ? e.right : e.left;  
        // ...  
    }  
}
```

Otherwise, we know `e.kind` = “operator”, as `e.kind` cannot be “number” since we are told `e` must contain `x`.

We then determine which operand contains `x`. The variable `x` only appears once in the AST, so it can only be in one of the operands.

We store variables to represent the operands with and without `x`.

Exercise 2: Solution

```
function solveForX(e: Expression, result: number): number {  
    if (e.kind === "variable") {  
        return result;  
    } else {  
        const leftHasX = e.left.containsX;  
        const sideWithX = leftHasX ? e.left : e.right;  
        const sideWithoutX = leftHasX ? e.right : e.left;  
        const solved = evalExpression(sideWithoutX);  
        // ...  
    }  
}
```

Next, we want to evaluate the operand that does not contain x. We can do so since this operand is guaranteed to be all numbers and binary operators, so we can calculate exactly what its value is.

We use the given evalExpression function to do this.

Exercise 2: Solution

```
function solveForX(e: Expression, result: number): number {  
    if (e.kind === "variable") {  
        return result;  
    } else {  
        const leftHasX = e.left.containsX;  
        const sideWithX = leftHasX ? e.left : e.right;  
        const sideWithoutX = leftHasX ? e.right : e.left;  
        const solved = evalExpression(sideWithoutX);  
        let new_result = ?  
        return solveForX(sideWithX, new_result);  
    }  
}
```

Now, notice that we could call `solveForX` recursively on `sideWithX` if we knew what the expected result of `sideWithX` was. This is because `sideWithX` is an `Expression` containing the variable `x`.

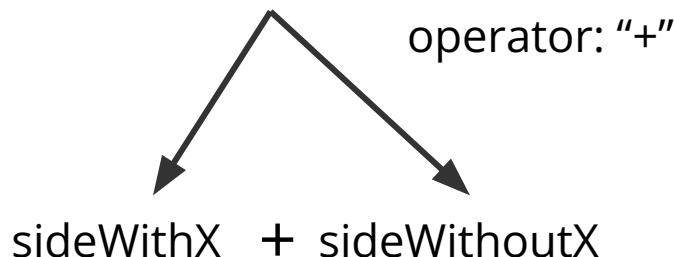
In fact, we can solve for what this expected result should be using what we know!

Exercise 2: Solution

Suppose $e.operator$ was "+". Then, we are expecting that

- $\text{value}(\text{sideWithX}) + \text{value}(\text{sideWithoutX}) = \text{result}$
- This means:
 - $\text{value}(\text{sideWithX}) = \text{result} - \text{value}(\text{sideWithoutX})$

e : Expression



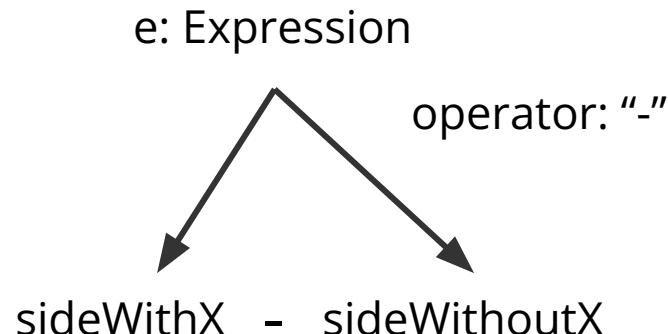
Notice that the ordering doesn't matter here;
 sideWithX could be on the right instead but we
still get

$$\text{value}(\text{sideWithX}) = \text{result} - \text{value}(\text{sideWithoutX})$$

Exercise 2: Solution

Now, what if $e.operator$ was “-”? Here, order matters.

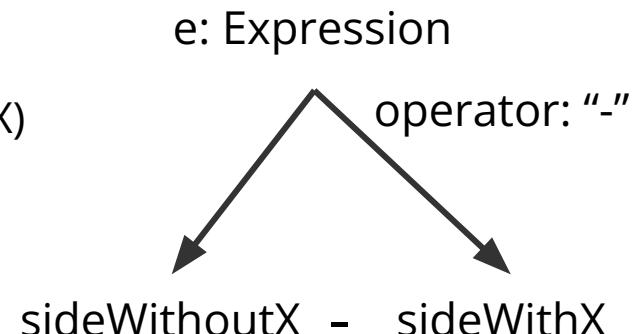
- If $leftHasX$, then we are expecting
 - $\text{value}(\text{sideWithX}) - \text{value}(\text{sideWithoutX}) = \text{result}$
 - This means:
 - $\text{value}(\text{sideWithX}) = \text{result} + \text{value}(\text{sideWithoutX})$



Exercise 2: Solution

Now, what if $e.operator$ was “-”? Here, order matters.

- If $leftHasX$, then we are expecting
 - $\text{value(sideWithX)} - \text{value(sideWithoutX)} = \text{result}$
 - This means:
 - $\text{value(sideWithX)} = \text{result} + \text{value(sideWithoutX)}$
- Otherwise, x is on the right side, so we are expecting
 - $\text{value(sideWithoutX)} - \text{value(sideWithX)} = \text{result}$
 - This means:
 - $\text{value(sideWithX)} = \text{value(sideWithoutX)} - \text{result}$



Exercise 2: Solution

```
function solveForX(e: Expression, result: number): number {  
    if (e.kind === "variable") {  
        return result;  
    } else {  
        const leftHasX = e.left.containsX;  
        const sideWithX = leftHasX ? e.left : e.right;  
        const sideWithoutX = leftHasX ? e.right : e.left;  
        const solved = evalExpression(sideWithoutX);  
        let new_result = 0;  
        if (e.operator === "+") {  
            new_result = result - solved;  
        } else if (leftHasX) {  
            new_result = result + solved;  
        } else {  
            new_result = solved - result;  
        }  
        return solveForX(sideWithX, new_result);  
    }  
}
```

We implement the logic from the previous slide with the if statements on the left.

Note that this could be simplified or done in other ways as well (i.e. using ternaries).

This allows us to obtain the new_result we expect for sideWithX.

We can then call solveForX recursively!

Exercise 2: Solution

```
function solveForX(e: Expression, result: number): number {
    switch (e.kind) {
        case "number":
            return e.value;
        case "variable":
            return result;
        default:
            if (!e.containsX) {
                const [left, right] = [solveForX(e.left, 0), solveForX(e.right, 0)];
                return e.operator === "+" ? left + right : left - right;
            }
            const leftHasX = e.left.containsX
            const solved = leftHasX ? solveForX(e.right, 0) : solveForX(e.left, 0);
            const toSolve = leftHasX ? e.left : e.right;
            let expected = result - solved;
            if (e.operator === "-") {
                expected = leftHasX ? result + solved : solved - result;
            }
            return solveForX(toSolve, expected);
    }
}
```

As a bonus, here is how you could implement this without evalExpression.

Anonymous Lab Feedback

UMassAmherst

Manning College of Information
& Computer Sciences



<https://forms.gle/iciZEVcuhF4UDNzQ6>