

UMassAmherst

Manning College of Information  
& Computer Sciences

Programming Methodology

## Lab 12: Interpreters

Wednesday, May 1st, 2024



# Weekly Lab Agenda

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

# Reminders

- Homework 8 (interpreter) will be posted and due Thursday 5/9 EOD
- HW 7 is due tomorrow EOD
- HW7 CATME feedback form is due next Monday 5/6 EOD
- HW 7 Self Reflection is due Friday 5/10 EOD
  - More info will be posted soon

# Today's Goals

- Practice working with interpreter concepts

# Interpreters.


An interpreter is a program that runs programs.

**Parser** takes a source program (concrete syntax) and turns it into an abstract syntax tree

**Grammar** describes the structure of a correct program.

a grammar is a set of rules that determine the action that the parser should perform

here is an example of grammar  
(the grammar for hw8)



UMassAmherst

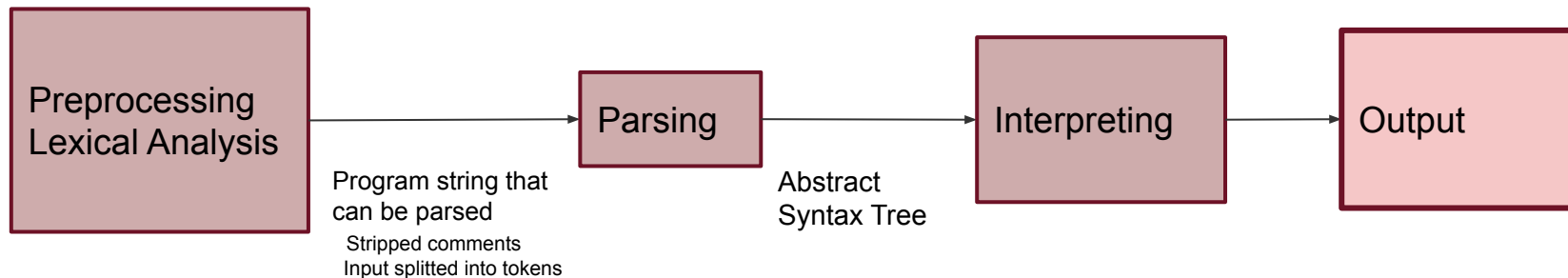
Manning College of Information  
& Computer Sciences

Numbers	$n ::= \dots$	numeric (positive and negative integer numbers)
Variables	$x ::= \dots$	variable name (a sequence of uppercase or lowercase alphabetic letters)
Expressions	$e ::=$   $n$   $\text{true}$   $\text{false}$   $x$   $e_1 + e_2$   $e_1 - e_2$   $e_1 * e_2$   $e_1 / e_2$   $e_1 \ \&\& \ e_2$   $e_1 \    \ e_2$   $e_1 < e_2$   $e_1 > e_2$   $e_1 == e_2$	numeric constant boolean value true boolean value false variable reference addition subtraction multiplication division logical and logical or less than greater than equal to
Statements	$s ::=$   $\text{let } x = e;$   $x = e;$   $\text{if } (e) \ b_1 \ \text{else } b_2$   $\text{while } (e) \ b$   $\text{print}(e);$	variable declaration assignment conditional loop display to console
Blocks	$b ::= \{ s_1 \dots s_n \}$	
Programs	$p ::= s_1 \dots s_n$	

# Interpreters.

UMassAmherst

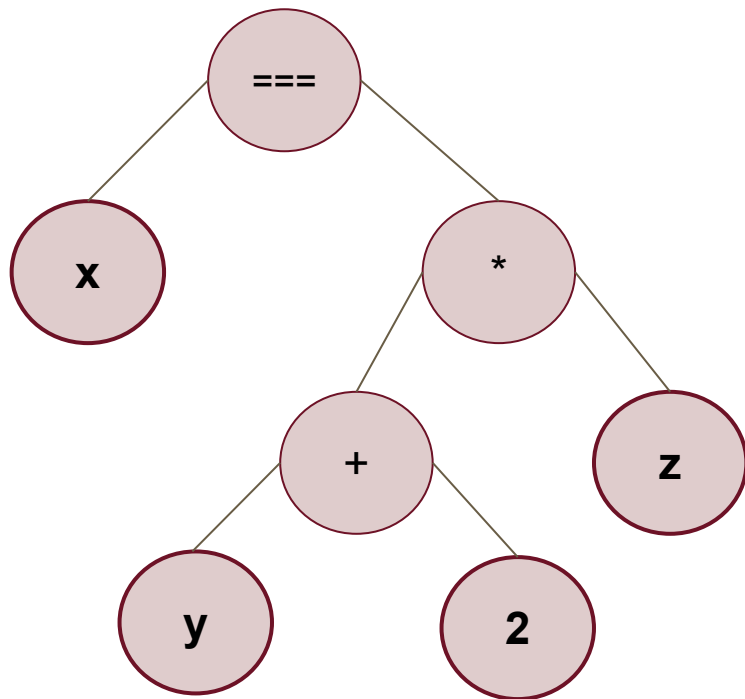
Manning College of Information  
& Computer Sciences



# Expression Evaluation

What would the AST of this expression look like?

**$x === (y + 2) * z$**

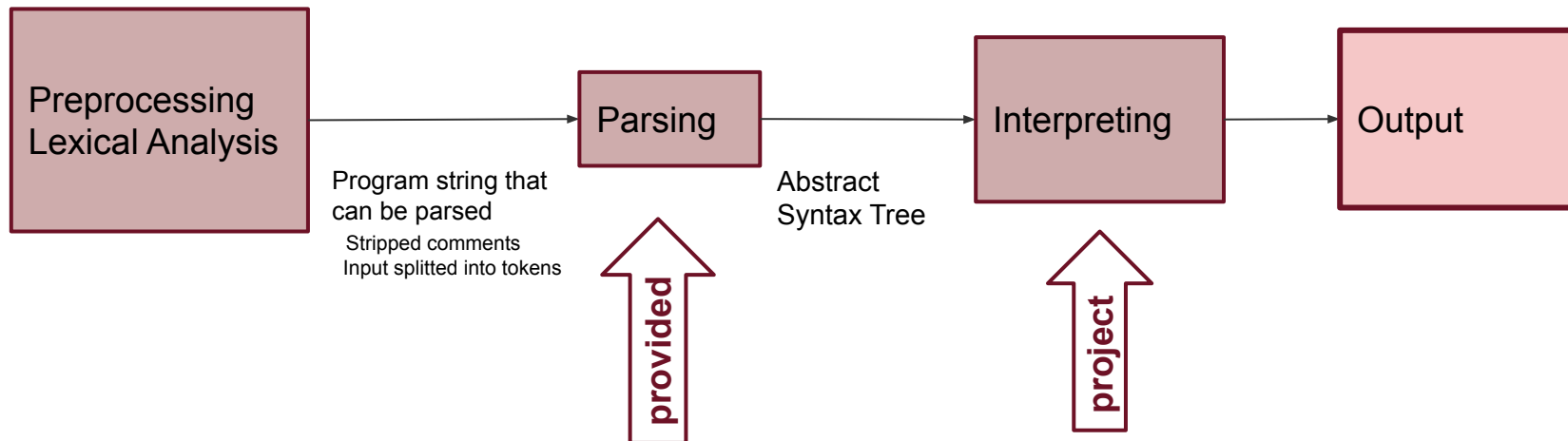


UMassAmherst

Manning College of Information  
& Computer Sciences

```
{
  kind: "operator",
  operator: "===",
  left: {
    kind: "variable",
    name: "x"
  },
  right: {
    kind: "operator",
    operator: "*",
    left: {
      kind: "operator",
      operator: "+",
      left: {
        kind: "variable",
        name: "y"
      },
      right: {
        kind: "number",
        value: 2
      }
    },
    right: {
      kind: "variable",
      name: "z"
    }
  }
}
```

# Interpreters.



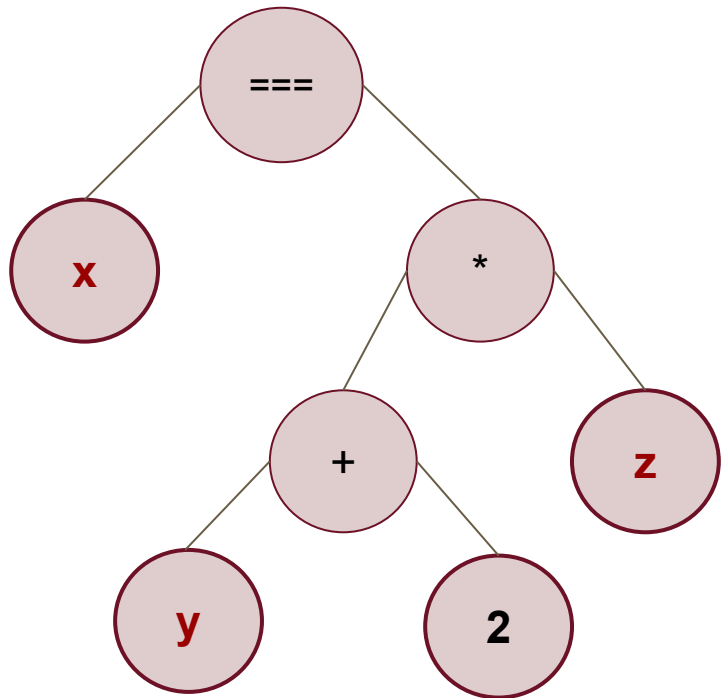
parsing functions provided for the homework:

- **parseExpression** parses an expression (e)
- **parseProgram** parses a program (p)



# Expression Evaluation

$x === (y + 2) * z$



Value of the variables will come from the **state**.

UMassAmherst

Manning College of Information  
& Computer Sciences

```
{
  kind: "operator",
  operator: "===",
  left: {
    kind: "variable",
    name: "x"
  },
  right: {
    kind: "operator",
    operator: "*",
    left: {
      kind: "operator",
      operator: "+",
      left: {
        kind: "variable",
        name: "y"
      },
      right: {
        kind: "number",
        value: 2
      }
    },
    right: {
      kind: "variable",
      name: "z"
    }
  }
}
```

# Constant Folding

Constant Folding refers to the process of evaluating constant subexpressions within a program.

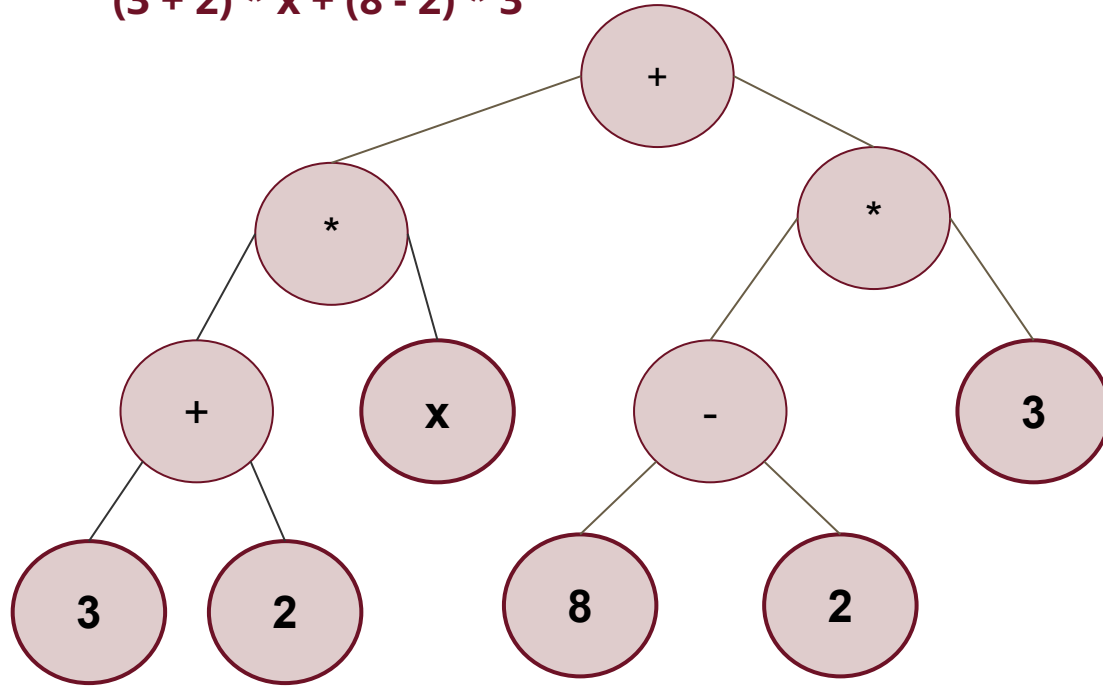
You can evaluate part of the expression without knowing variable values.

# Example: Constant Folding

UMassAmherst

Manning College of Information  
& Computer Sciences

$$(3 + 2) * x + (8 - 2) * 3$$

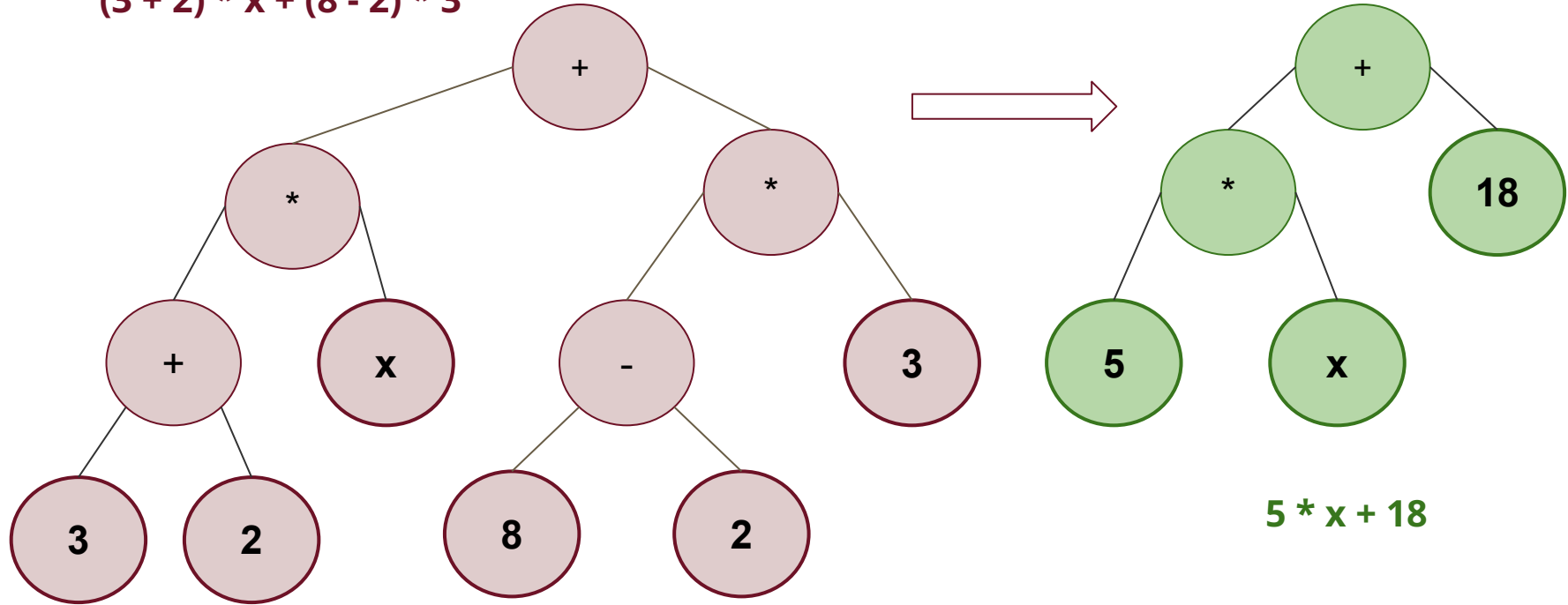


# Example: Constant Folding

UMassAmherst

Manning College of Information  
& Computer Sciences

$$(3 + 2) * x + (8 - 2) * 3$$



# Exercise: Type Inference

Implement a function to infer variable types from an expression containing binary operators. Assume that the expression tree is valid, but check for type mismatches. If there is a mismatch, throw an error.

*Notes:*

- *Variables may only be number or boolean*
- *Operators (+, -, \*, /, >, <, ==, &&, || ) are binary*
- *there is no constraint on the operand types of '=='*

*Example 1:*

- for “x + 2” output “number”; env object is: { x: “number” }

*Example 2:*

- for “x == y + z + 1” output “boolean”; env is: { x: “any”, y: “number”, z: “number” }

Infer types top-down: for each operator, you know its result type and the operand types  
When encountering a variable, check the required type is the same as the type stored in environment (when first encountered, store its inferred type)

## Solution: Type Inference

```
function typeCheck(e: Expression, expected: Type, env: TypeMap): Type {
  function checkBoth(e: BinExp, operandType: Type, resultType: Type) {
    typeCheck(e.left, operandType, env);
    typeCheck(e.right, operandType, env);
    checkEq(e.operator, resultType, expected);
    return resultType;
  }

  switch(e.kind) {
    case 'boolean':
    case 'number': checkEq(e.value, e.kind, expected);
                  return e.kind;
    case 'variable':
      if (!(e.name in env) || env[e.name] === `any`) {
        env[e.name] = expected
      }
      checkEq(e.name, env[e.name], expected);
      return env[e.name];
    case 'operator':
      return boolOp(e.operator) ? checkBoth(e, 'boolean', 'boolean')
        : cmpOp(e.operator) ? checkBoth(e, 'number', 'boolean')
        : mathOp(e.operator) ? checkBoth(e, 'number', 'number')
        : e.operator === '===' ? checkBoth(e, 'any', 'boolean')
        : (() => { throw new Error(`${e.operator} is not a valid operator.`); })();
    default: return expected;
  }
}
```

# Looking ahead

The next slides and exercise 2 are here for you to explore on your own later this week when you see the content in lecture.

The full autograder test suite is available in the starter code, so you can play around with the code on your own. Solutions will be posted.

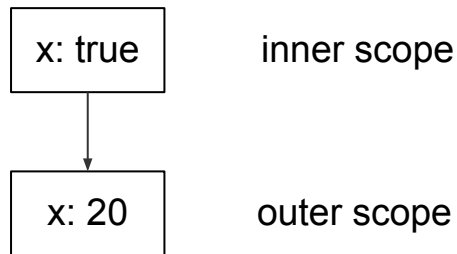
# Scoping review

A **block** introduces a new **inner** scope.

- a variable declared in an inner scope is not accessible after exiting the scope
- a variable declared in an inner scope can shadow a variable declared in an outer scope.

A scope will be represented by a **state**, which holds information of variable values.

```
let x = 10;  
if (x > 0) {  
  x = 20;  
  let x = true;  
}
```





## Exercise 2: Scoping

Implement a function, **printDecls**, that traverses a program's **Abstract Syntax Tree (AST)** and when reaching the end of each scope, prints all variables that were declared in that scope.

- Check that there are **no duplicate** declarations within a scope.
  - Print exactly: `duplicate declaration: \${variable name here}`
- Prefix each variable with the nesting level of the scope (the global scope is level 0).
  - Print exactly: `\${nesting level here}: \${variable name here}`

You can use any representation of scopes you like.

e.g. place variable names in an array or make them properties of an object.

You can use an array of scopes, adding a new scope when entering each block, or a linked list whose elements are scopes, or just have a local scope object when handling each block.

# Solution

```
import { Statement, parseProgram } from "../include/parser.js";

function printDecls(program: Statement[], level: number) {
  function doDecl(s: Statement, scope: {[key: string]: any}) {
    switch(s.kind) {
      case "let": {
        if (s.name in scope) { // or: scope.hasOwnProperty(name)
          console.log("duplicate declaration: " + s.name);
        } else {
          Object.defineProperty(scope, s.name, {}); // scope[s.name] = {}
        }
        return;
      }
      case "if":
        printDecls(s.truePart, level + 1);
        printDecls(s.falsePart, level + 1);
        return;
      case "while":
        printDecls(s.body, level + 1);
        return;
      default:
        return;
    }
  }

  let scope = {};
  program.forEach(s => doDecl(s, scope));
  let names = Object.getOwnPropertyNames(scope); // or Object.keys(scope);
  names.forEach(n => console.log(level.toString() + ": " + n));
}
```

UMassAmherst

Manning College of Information  
& Computer Sciences

```
// Test
const program = parseProgram(`
  let x = 1;
  let y = 2;
  x2 = 5;
  print(x2);

  if (x > y) {
    let z = 3;
    if (x > z) {
      let x = 4;
    } else {
      let t = 5;
    }
  } else {
    let p = 6;
  }

  while (true) {
    let w = 7;
  }
`);

printDecls(program, 0);

/**
Expected output:
2: x
2: t
1: z
1: p
1: w
0: x
0: y
**/
```

# Alternate Solution

UMassAmherst

Manning College of Information  
& Computer Sciences

```
import { Statement, parseProgram } from
"../include/parser.js";

function printDecls(program: Statement[], level: number) {
  function printDeclsHelper(s: Statement, level: number): void {
    switch (s.kind) {
      case "let":
        if (level in scope && s.name in scope[level]) {
          console.log( "Duplicate declaration " + s.name);
        } else {
          level in scope ? scope[level].push(s.name) :
scope[level] = [s.name]
        }
        break;
      case "if":
        (s.truePart).forEach(s => printDeclsHelper(s, level+ 1));
        (s.falsePart).forEach(s => printDeclsHelper(s, level+ 1));
        break;
      case "while":
        (s.body).forEach(s=>printDeclsHelper(s, level+ 1));
        break;
      default:
        break;
    }
  }

  let scope : {[key: number]: string[]} = {};
  program.forEach(s => printDeclsHelper(s, 0));
  let names = Object.keys(scope).reverse();
  names.forEach((n: string) => scope[Number(n)].forEach(v =>
console.log(n + " : " + v)));
}
```

```
// Test
const program = parseProgram(`
  let x = 1;
  let y = 2;
  x2 = 5;
  print(x2);

  if (x > y) {
    let z = 3;
    if (x > z) {
      let x = 4;
    } else {
      let t = 5;
    }
  } else {
    let p = 6;
  }

  while (true) {
    let w = 7;
  }
`);

printDecls(program, 0);

/**
Expected output:
2: x
2: t
1: z
1: p
1: w
0: x
0: y
**/
```