

UMassAmherst

Manning College of Information
& Computer Sciences

Programming Methodology

Lab 12: Interpreters

Wednesday, April 30, 2025



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 9 (interpreter) is posted and due Thursday 5/8 EOD
- HW 8 Coding part is due tonight
 - One submission per team, add other members on gradescope
- HW 8 CATME Survey is due Monday May 5th
- HW 8 Self Reflection is due Friday May 9th
 - More info will be posted soon

Today's Goals

- Practice working with interpreter concepts
- Combining async and interpreters

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

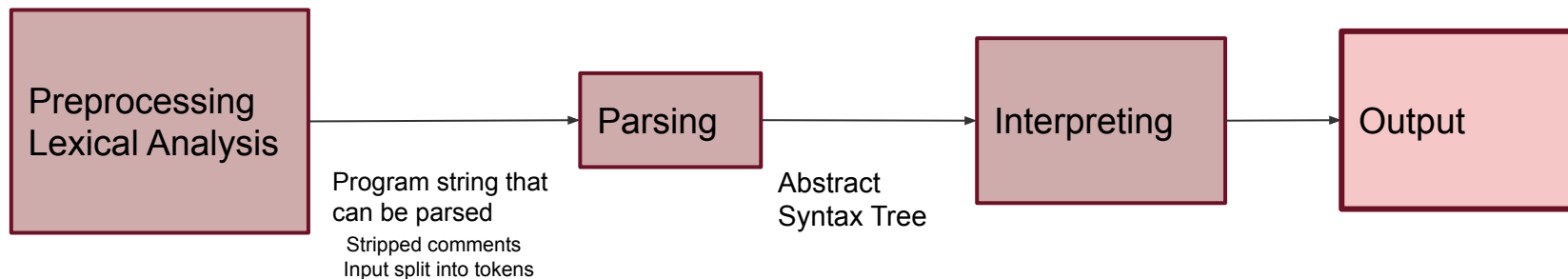
Manning College of Information
& Computer Sciences

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 true 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	$s ::=$ $\text{let } x = e;$ $x = e;$ $\text{if } (e) \text{ } b_1 \text{ else } b_2$ $\text{while } (e) \text{ } b$ $\text{print}(e);$	variable declaration assignment conditional loop display to console
Blocks	$b ::= \{ s_1 \dots s_n \}$	
Programs	$p ::= s_1 \dots s_n$	

Interpreters.

UMassAmherst

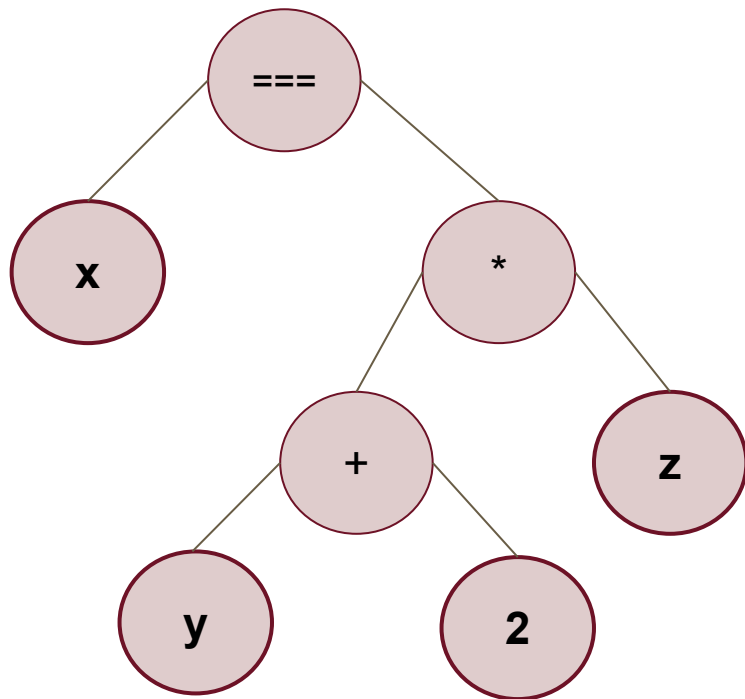
Manning College of Information
& Computer Sciences



Expression Evaluation

What would the AST of this expression look like?

x === (y + 2) * z

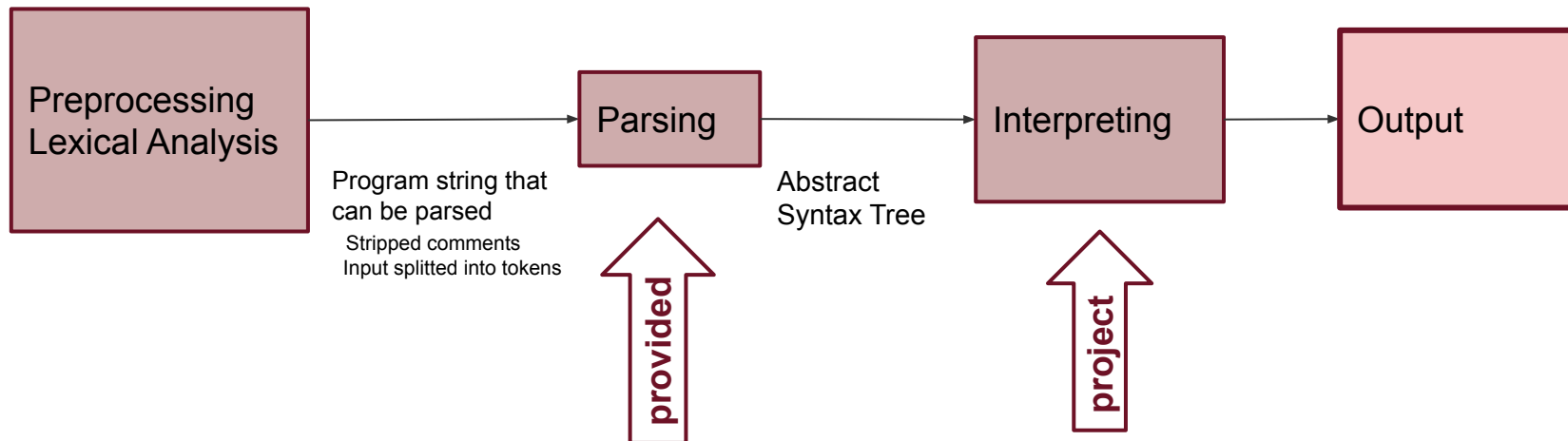


UMassAmherst

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.

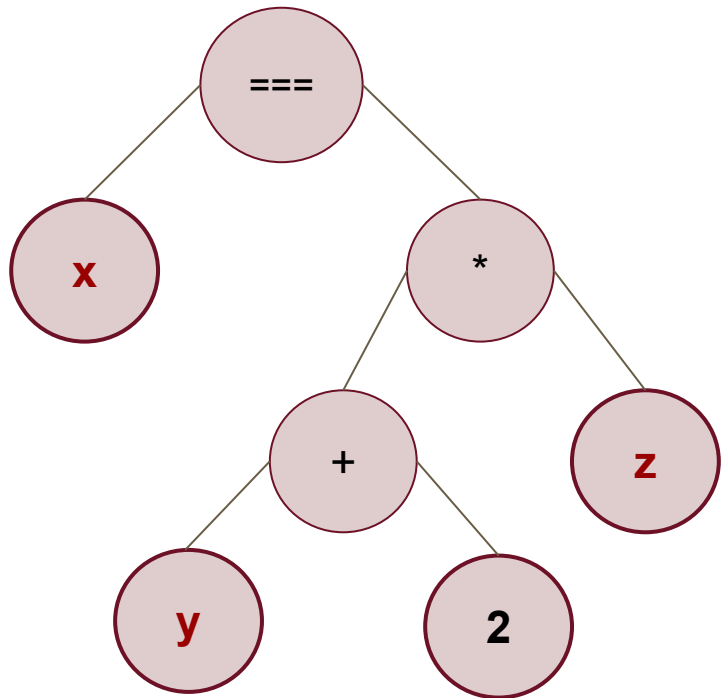


parsing functions provided for the homework:

- **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"
    }
  }
}
```

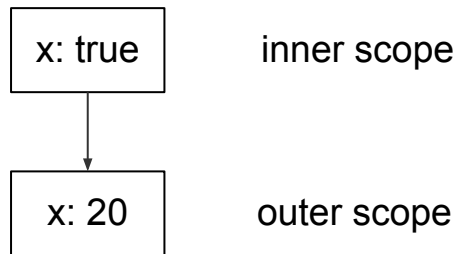
Scoping review

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 1: Scoping

Implement a function **printDecls** that traverses a program's **Abstract Syntax Tree (AST)** and at 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: `duplicate declaration: \${variable name here}`
- Prefix each variable with the nesting level of the scope (the global scope is level 0).
 - Print: `\${nesting level here} : \${variable name here}`

Use any representation of scopes you like: array of variable names, `Set<string>`, or make variables properties of an object. There is no need to link scopes for this task.

Solution

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

function printDecls(block: Statement[], level: number) {
  const scope: Record<string, any> = {};
  for (const s of block)
    switch(s.kind) {
      case "let":
        // `in` also follows prototype chain but we have none here
        if (scope.hasOwnProperty(s.name))
          console.log("duplicate declaration: " + s.name);
        else scope[s.name] = 1; // anything, really
        break;
      case "if":
        printDecls(s.truePart, level + 1);
        printDecls(s.falsePart, level + 1);
        break;
      case "while":
        printDecls(s.body, level + 1);
        break;
    }
  for (const name in scope) // iterates over keys
    // console.log(level, ":", name); Will print a space on both sides
    console.log(String(level) + ": " + name);
  // or use Object.keys(scope) or Object.getOwnPropertyNames(scope)
}
```

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
**/
```

Exercise 2

Write a function `interpExpressionAsync(s: State, e: Expression):`

`Promise<boolean>` that evaluates a boolean expression in a global state. The relevant types are:

```
type State = { [key: string]: boolean }
```

```
type AsyncExpression = { kind: "boolean"; value: boolean } | { kind:  
"variable"; name: string } | { kind: "operator"; operator: "&&" | "||";  
left: AsyncExpression; right: AsyncExpression };
```

For a constant, fulfill with its value. For a variable, reject if the variable is not defined, else fulfill with the variable value. For a boolean operator, fulfill by using short-circuit evaluation of the fulfillment values, like in JavaScript, or propagate the rejection if any evaluated subexpression rejects

Exercise 2 Solution

UMassAmherst

Manning College of Information
& Computer Sciences

```
export function interpExpressionAsync(s: State, e: AsyncExpression): Promise<boolean>{
  switch(e.kind){
    case "boolean" : return Promise.resolve(e.value);
    case "variable" : return e.name in s ? Promise.resolve(s[e.name]): Promise.reject("undefined variable");
    case "operator" : return interpExpressionAsync(s, e.left).then( b1=>
      e.operator === "&&" ? b1 ? interpExpressionAsync(s, e.right) : false
      : b1 ? true : interpExpressionAsync(s, e.right));
  }
}
```

Take Home: Type Inference

Implement a function that infers variable types from an expression AST with binary operators. Check for type mismatches and throw an error if one is found.

- *As in our toy language, types may only be number or boolean*
- *Binary operators are +, -, *, /, >, <, ==, &&, ||*
- *There is no constraint on the operand types of '=='*

Expression 1: “x + 2” output “number”; env object is: { x: “number” }

Expression 2: “x == y + z + 1” output “boolean”; env is: { x: “any”, y: “number”, z: “number” }

Your recursive function should pass and update an environment object with variable types.

When encountering an operator, pass down an expected type for left and right operands.

When encountering a variable, check that the required type is the same as the type stored in the environment (when first encountered, store its inferred type)

Solution: Take Home 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;
  }
}
```