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.

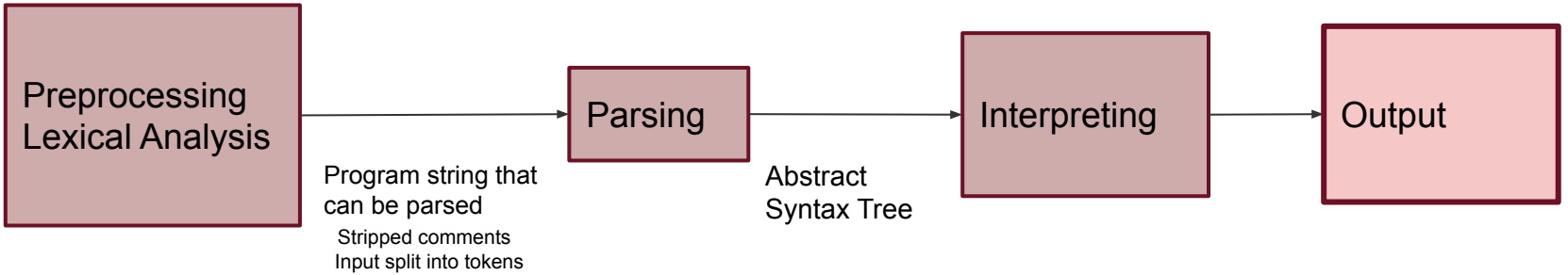
> a grammar is a set of rules that determine the action that the parser should perform

here is an example of grammar

(the grammar for hw8)

Numbers	n ::= ...	numeric (positive and negative integer numbers)
Variables	x ::= ...	variable name (a sequence of uppercase or lowercase alphabetic letters)
Expressions	e ::= n	numeric constant
	| true	boolean value true
	| false	boolean value false
	| x	variable reference
	| e1 + e2	addition
	| e1 - e2	subtraction
	| e1 * e2	multiplication
	| e1 / e2	division
	| e1 && e2	logical and
	| e1 || e2	logical or
	| e1 < e2	less than
	| e1 > e2	greater than
	| e1 === e2	equal to
Statements	s ::= let x = e;	variable declaration
	| x = e;	assignment
	| if (e) b1 else b2	conditional
	| while (e) b	loop
	| print(e);	display to console
Blocks	b ::= { s1 ... sn }	
Programs	p ::= s1 ... sn	

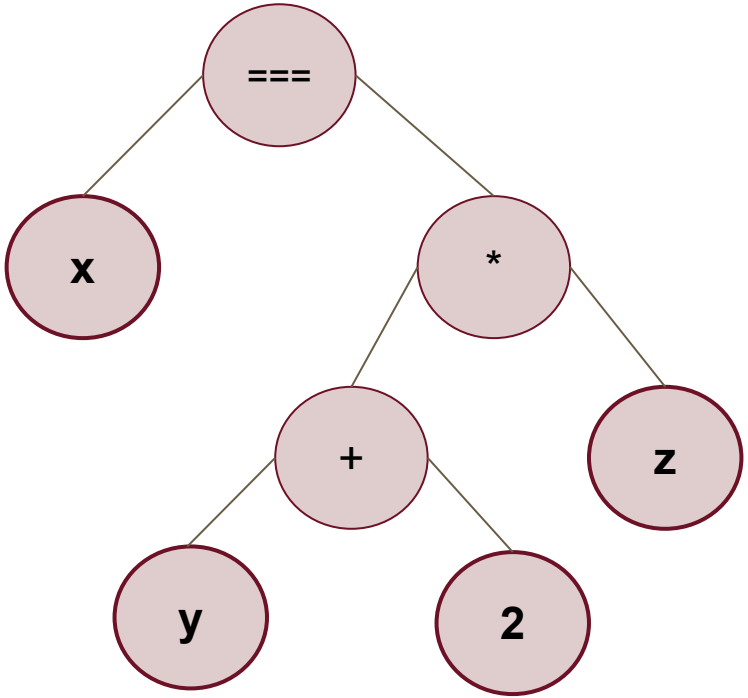
Interpreters.



Expression Evaluation

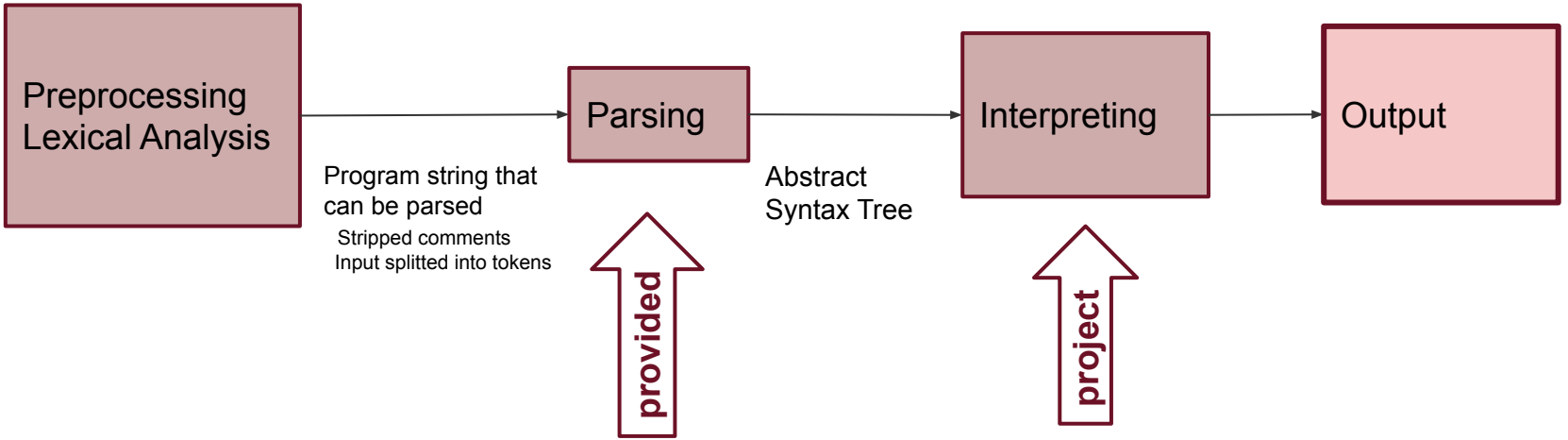
What would the AST of this expression look like?

x === (y + 2) * z




```
{
    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.

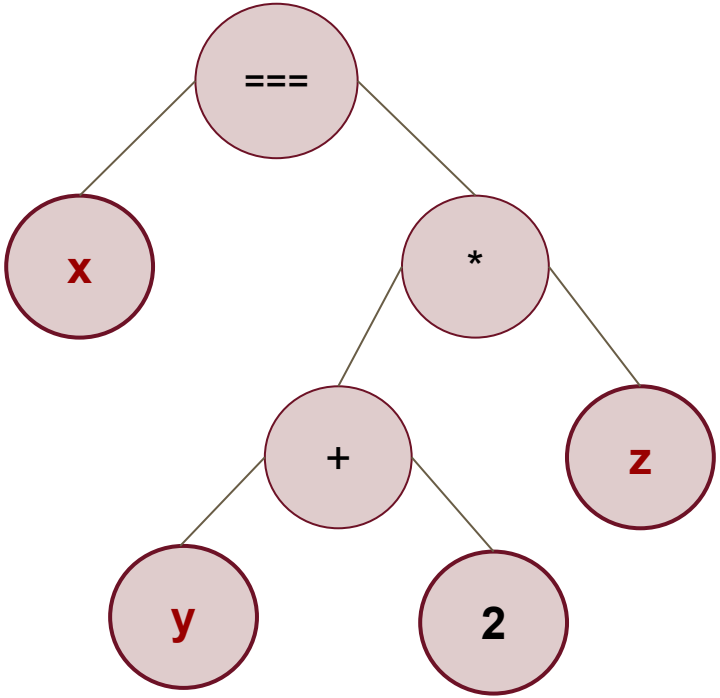


parsing functions provided for the homework:

- **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**.

```
{
    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"
      }
    }
}
```

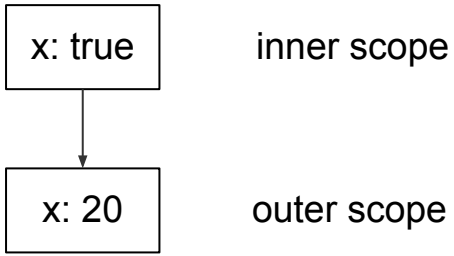
Scoping review

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.

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

From Fall 2023 Final Exam

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)