UMass Amherst | Manning College of Information & Computer Sciences

Programming Methodology

Lab 13: Interpreters

Wednesday, December 4, 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) is posted and due Monday 12/9 EOD
- HW 7 CATME Survey is due today EOD
- HW 7 Self Reflection is due Wednesday 12/11 EOD
  - More info will be posted soon
- Final Exam is next Thursday (12/12)!
  - 3:30pm - 5:30pm in Totman Gym
- Final Exam Review Sessions
  - Max - Wednesday 12/11 3pm - 5pm @ ILC TBD
  - SI (Lucy) - Wednesday 12/11 7pm - 9pm @ Thompson 104
  - Both sessions will be recorded

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

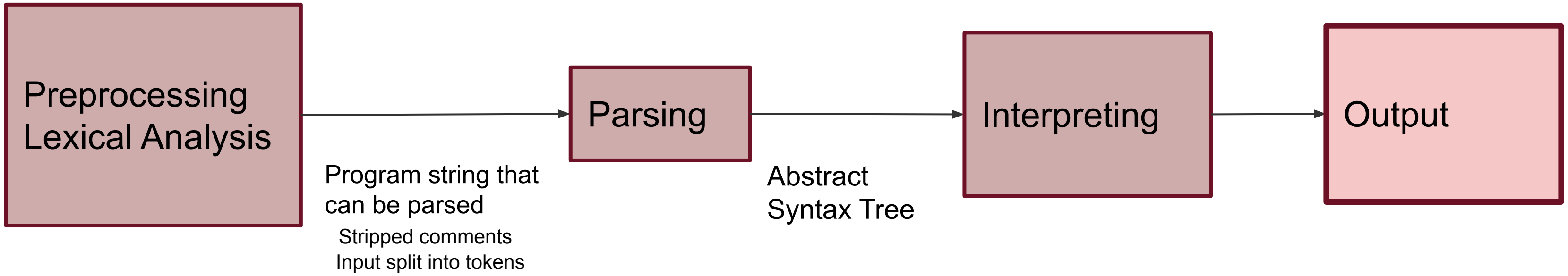
here is an example of grammar

(the grammar for hw8)

```
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.

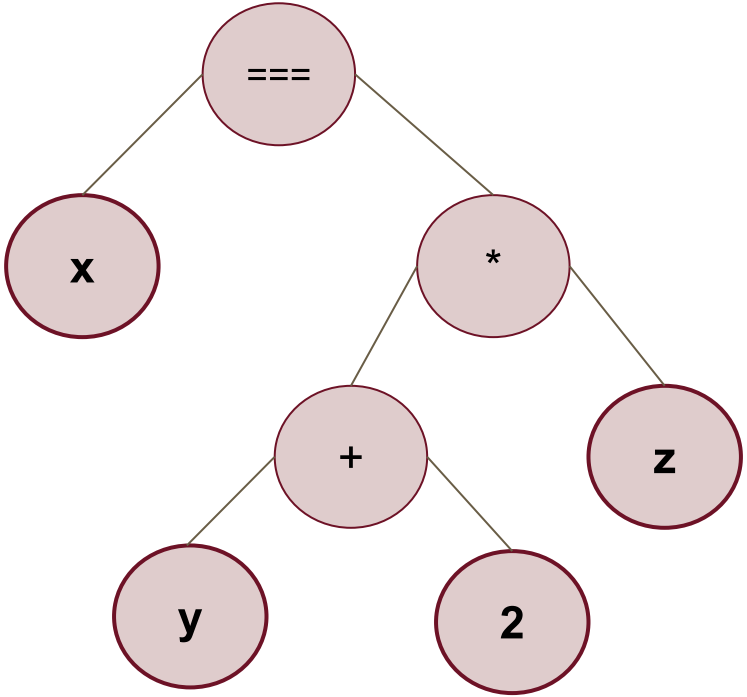
UMassAmherst | Manning College of Information & Computer Sciences



Expression Evaluation

What would the AST of this expression look like?

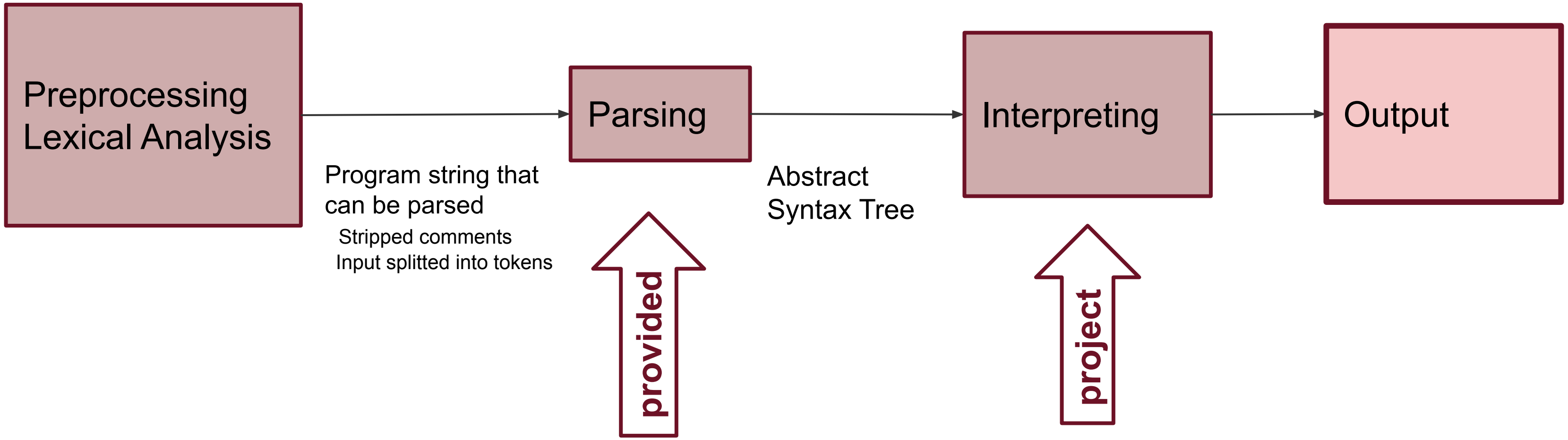
x === (y + 2) * z




```
{
    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.

UMassAmherst | Manning College of Information & Computer Sciences



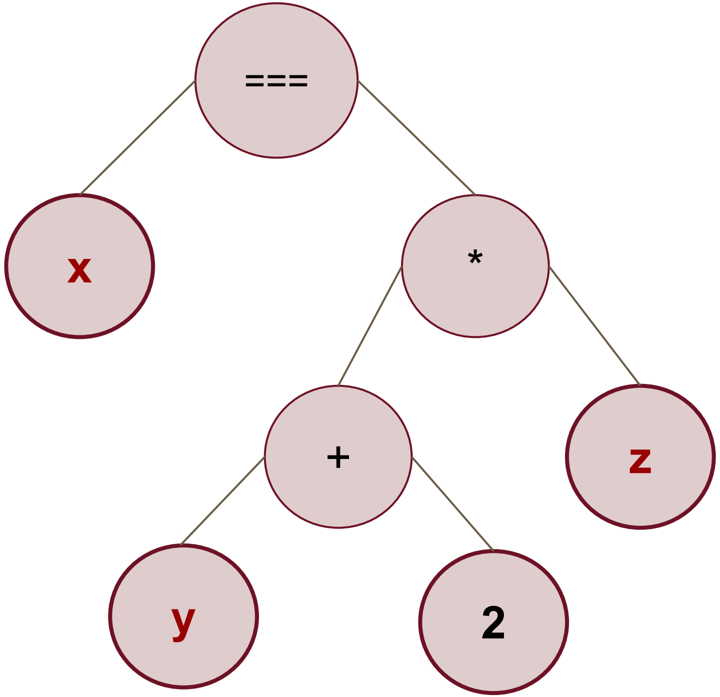
parsing functions provided for the homework:

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

Expression Evaluation

UMassAmherst | Manning College of Information & Computer Sciences

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"
      }
    }
}
```

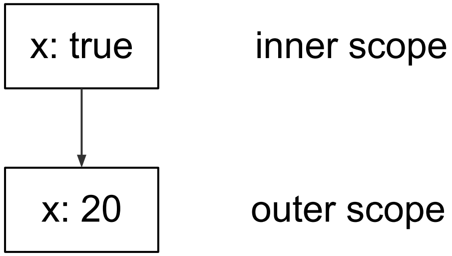
Scoping review

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

- a variable declared in an inner scope is <u>not accessible</u> after exiting the scope
- a variable declared in an inner scope can <u>shadow</u> 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

UMassAmherst | Manning College of Information & Computer Sciences

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

UMassAmherst | Manning College of Information & Computer Sciences

```
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)
}
```

```
// 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


```
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: 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: 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;
  }
}
```