CSC 416/565: DESIGN AND CONSTRUCTION OF COMPILERS

West Chester University
Dr. Richard Burns
Spring 2023

Data structures for tree languages

Programming Assignment #1 Introduction

“Straight Line Program Language” will have statements and expressions.

- no loops or if conditionals

Will specify language using a grammar.

A Grammar has:

- terminals (such as `id`)
- non-terminals (such as `Stm`)

```
(CompoundStm) Stm -> Stm ; Stm
(AssignStm)   Stm -> id := Exp
(PrintStm)    Stm -> print(ExpList)
(IdExp)       Exp -> id
(NumExp)      Exp -> num
(OpExp)       Exp -> Exp Binop Exp

(Plus)        Binop -> +
              Binop -> -
              Binop -> x
              Binop -> /

(PairExpList) ExpList -> Exp , ExpList
(LastExpList) ExpList -> Exp

(EseqExp)     Exp -> (Stm, Exp)   // Expression Sequence
```

Expression Sequences

```
int a = 1, b = 2;
int i;
i = (a += 2, a + b);
```

What is the answer?

In general:

- Statements can have side-effects, but they don't return a value.
- Expressions return a value.

Example Source Program in Tree Language

```
a := 5 + 3 ; b := (print(a, a-1), 10 * a) ; print(b)
```

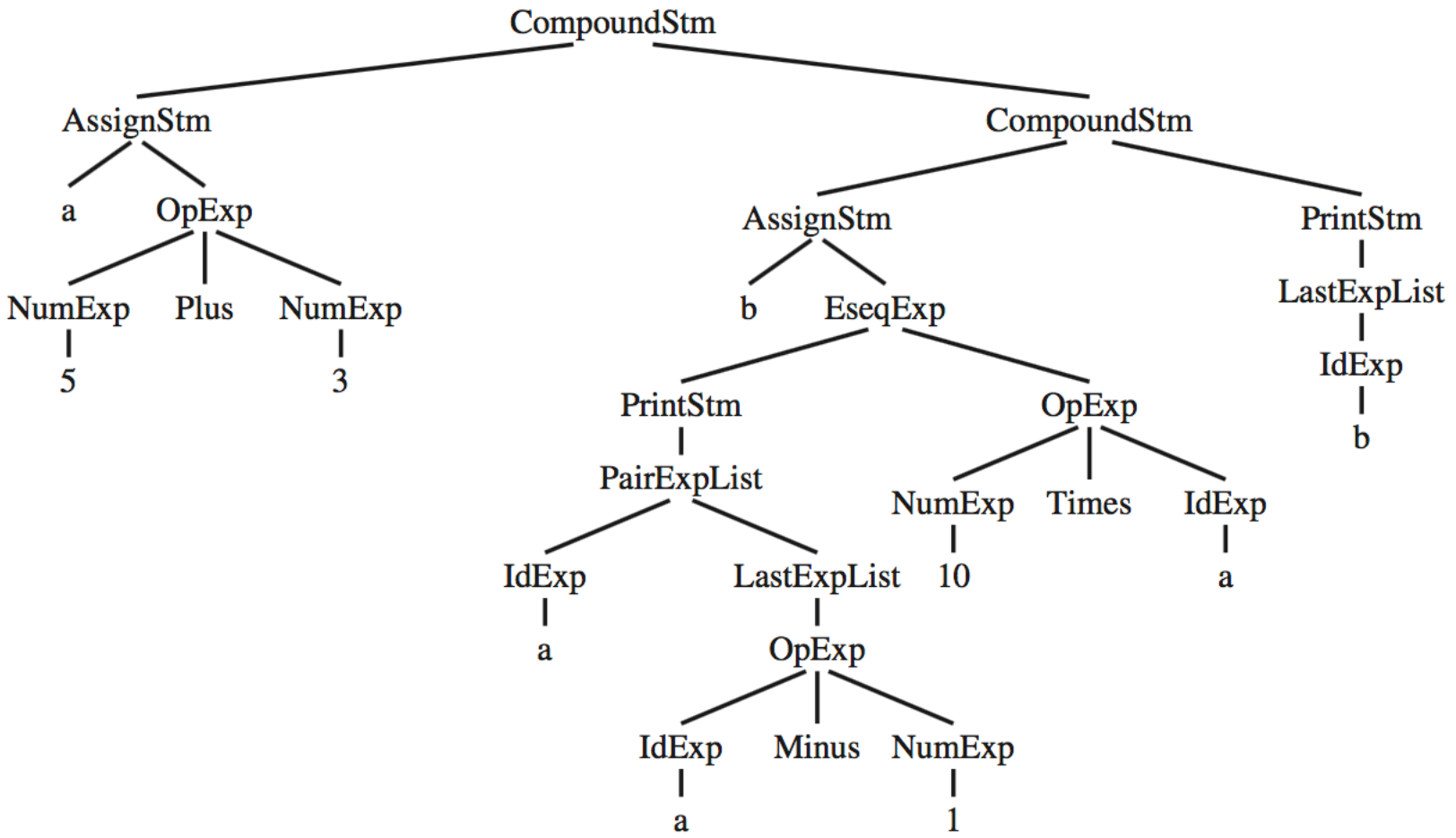
What is the output?

Programming Assignment #1: Write an *Interpreter* for the Tree Language

- Going to skip *scanning* and *parsing* and will go right into an IR using a tree data structure
- Although our grammar is *ambiguous*, this is not a problem for this assignment.

Ambiguous?

- There are two valid parse tree derivations using our grammar for the expression: `4 * 3 + 5`




```
a := 5 + 3 ; b := ( print ( a , a - 1 ) , 10 * a ) ; print ( b )
```

Appel, Figure 1.4

```
public abstract class Stm {}

public class CompoundStm extends Stm {
  public Stm stm1, stm2;
  public CompoundStm(Stm s1, Stm s2) {stm1=s1; stm2=s2;}
}

public class AssignStm extends Stm {
  public String id;
  public Exp exp;
  public AssignStm(String i, Exp e) {id=i; exp=e;}
}

public class PrintStm extends Stm {
  public ExpList exps;
  public PrintStm(ExpList e) {exps=e;}
}
```

```
public abstract class Exp {}

public class IdExp extends Exp {
  public String id;
  public IdExp(String i) {id=i;}
}

public class NumExp extends Exp {
  public int num;
  public NumExp(int n) {num=n;}
}

public class OpExp extends Exp {
  public Exp left, right;
  public int oper;
  final public static int Plus=1,Minus=2,Times=3,Div=4;
  public OpExp(Exp l, int o, Exp r) {left=l; oper=o; right=r;}
}

public class EseqExp extends Exp {
  public Stm stm;
  public Exp exp;
  public EseqExp(Stm s, Exp e) {stm=s; exp=e;}
}
```

```
public abstract class ExpList {}

public class PairExpList extends ExpList {
  public Exp head;
  public ExpList tail;
  public PairExpList(Exp h, ExpList t) {head=h; tail=t;}
}

public class LastExpList extends ExpList {
  public Exp head;
  public LastExpList(Exp h) {head=h;}
}
```

The Task:

Write single line programs, like example on page 12 [Appel].

Code two functions:

1. `int maxargs (Stm s)` - returns the maximum number of args in any print statement
2. `void interp (Stm s)` - interprets the program

Two ways to program:

1. Functionally using Java's `instanceof` operator
2. OO, by adding methods to each class and utilizing polymorphism

Strategies for implementing the interpreter found on [Appel], page 13.