ASTS, GRAMMARS, PARSING, TREE TRAVERSALS

Lecture 13

CS2110 – Spring 2017

Prelim 1 tonight!

Goldw

Last name A..V

SNOW DAY!

Keep an eye out for rescheduling details.

- Yo … y know which time slot you're taking the exam.
- **Brin … r Cornell ID**. You will need it to get into the exam room.
- No recitation this week. **Tuesday** recitations are office hours open to all unless otherwise noted on Piazza.

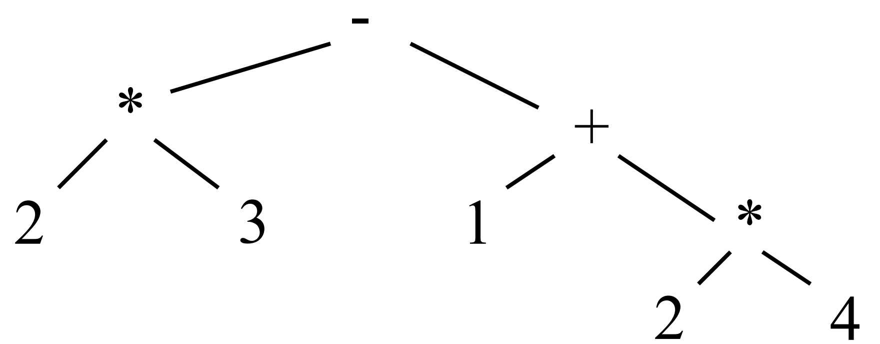
Today: Parse Trees, Parsing, and Grammars

- Parse trees: text, section 23.36
- Definition of Java Language, sometimes useful:
  docs.oracle.com/javase/specs/jls/se8/html/index.html
- Grammar for most of Java, for those who are curious:
  docs.oracle.com/javase/specs/jls/se8/html/jls-19.html
- Tree traversals –preorder, inorder, postorder: text, sections 23.13 .. 23.15.

Expression trees

From last time: we can draw a syntax tree for

2 * 3 – (1 + 2 * 4)




```
interface Expr {
    /* evalute this Expr and return the value*/
    public abstract int eval();
    /* return an infix representation */
    public abstract String infix();
}
```

Tree traversals

"Walking" over the whole tree is a tree traversal

- Done often enough that there are standard names

Previous example: in-order traversal

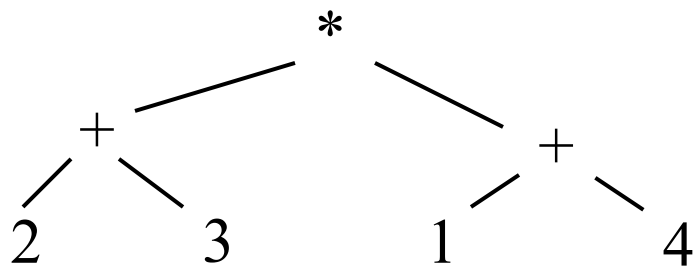
- Process left subtree
- Process root
- Process right subtree

Note: Can do other processing besides printing

Other standard kinds of traversals

- preorder traversal
  - Process root
  - Process left subtree
  - Process right subtree
- postorder traversal
  - Process left subtree
  - Process right subtree
  - Process root
- level-order traversal
  - Not recursive: uses a queue (we'll cover this later)

tree for (2 + 3) * (1 + 4)



Preorder traversal:
1. Visit the root
2. Visit left subtree, in preorder
3. Visit right subtree, in preorder

prefix and postfix notation proposed by Jan Lukasiewicz in 1951

$* \; + \; 2 \; 3 \quad + \; 1 \; 4$

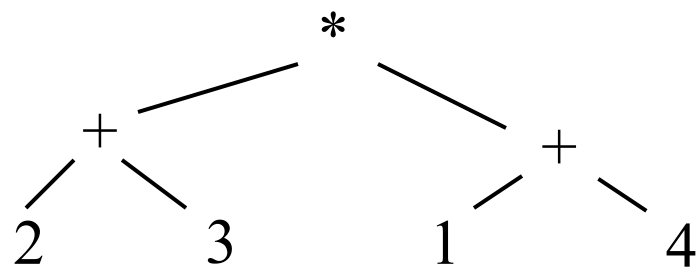
tree for (2 + 3) * (1 + 4)

Postfix is easy to compute. Process elements left to right.

Number? Push it on a stack

Binary operator? Remove two top stack elements, apply operator to it, push result on stack

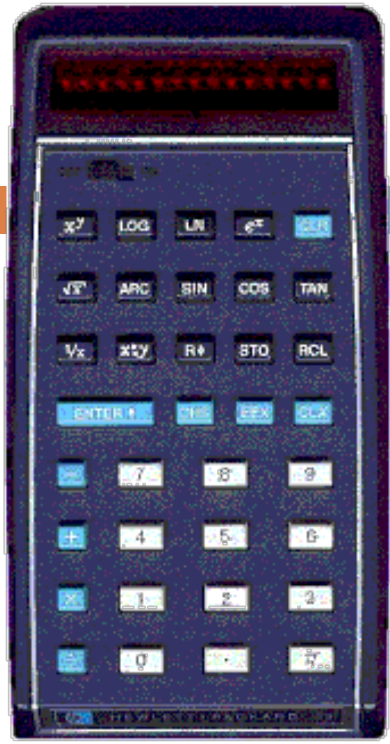
Unary operator? Remove top stack element, apply operator to it, push result on stack

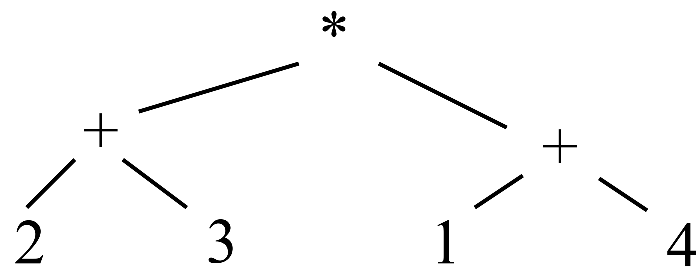


Postfix notation

2 3 + 1 4 + *

tree for (2 + 3) * (1 + 4)





In about 1974, Gries paid $300 for an HP calculator, which had some memory and used postfix notation! Still works. Come up to see it.

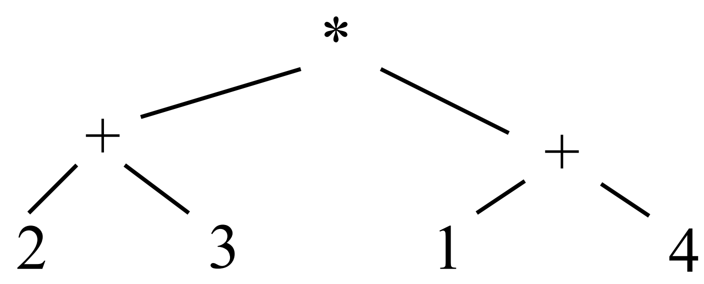
Postorder traversal:

1. Visit left subtree, in postorder
2. Visit right subtree, in postorder
3. Visit the root

2 3 + 1 4 + *

Postfix notation (also called Reverse Polish Notation)

tree for (2 + 3) * (1 + 4)



Inorder traversal:
1. Visit left subtree, in inorder
2. Visit the root
3. Visit right subtree, in inorder

To help out, put parens around expressions with operators

(2 + 3) * (1 + 4)

Prefix and Postfix Notation

Not as strange as it looks!

add(a, b) is prefix notation for the binary add operator!

(in some languages, this is simply written add a b)

n! is a postfix application of the factorial operator!

No parentheses needed!

Infix	Prefix	Postfix
(5 + 3) * 4	* + 5 3 4	5 3 + 4 *
5 + (3 * 4)	+ 5 * 3 4	5 3 4 * +
1+2+3*4-7	+ 1 + 2 - * 3 4 7	1 2 + 3 4 * + 7 -

Expression trees: in code


```
public interface Expr {
    String infix(); // returns an infix representation
    int eval(); // returns the value of the expression
}
```

```
public class Int implements Expr {
   private int v;
   public int eval() { return v; }
   public String infix() {
           return " " + v + " "; }
}
```

```
public class Sum implements Expr {
  private Expr left, right;
  public int eval() {
    return left.eval() + right.eval();
  }
  public String infix() {
    return "(" + left.infix() +
           "+" + right.infix() + ")";
  }
}
```

Motivation for grammars

- The cat ate the rat.
- The cat ate the rat slowly.
- The small cat ate the big rat slowly.
- The small cat ate the big rat on the mat slowly.
- The small cat that sat in the hat ate the big rat on the mat slowly, then got sick.
- …

- Not all sequences of words are legal sentences

  The ate cat rat the
- How many legal sentences are there?
- How many legal Java programs?
- How do we know what programs are legal?

http://docs.oracle.com/javase/specs/jls/se8/html/index.html

A Grammar

Sentence → Noun Verb Noun

Noun → goats

Noun → astrophysics

Noun → bunnies

Verb → like

| see

- White space between words does not matter
- A very boring grammar because the set of Sentences is finite (exactly 18 sentences)

Our sample grammar has these rules:

A Sentence can be a Noun followed by a Verb followed by a Noun

A Noun can be goats or astrophysics or bunnies

A Verb can be like or see

A Grammar

Sentence → Noun Verb Noun
Noun → goats
Noun → astrophysics
Noun → bunnies
Verb → like
Verb → see

Grammar: set of rules for generating sentences of a language.

Examples of Sentence:

- goats see bunnies
- bunnies like astrophysics

- The words goats, astrophysics, bunnies, like, see are called *tokens* or *terminals*
- The words Sentence, Noun, Verb are called *nonterminals*

A recursive grammar

Sentence → Sentence and Sentence
Sentence → Sentence or Sentence
Sentence → Noun Verb Noun
Noun → goats
Noun → astrophysics
Noun → bunnies
Verb → like
| see

This grammar is more interesting than previous one because the set of Sentences is infinite

What makes this set infinite?
Answer:
Recursive definition of Sentence

Aside

What if we want to add a period at the end of every sentence?

Sentence → Sentence and Sentence .

Sentence → Sentence or Sentence .

Sentence → Noun Verb Noun .

Noun → …

Does this work?

No! This produces sentences like:

$$\underbrace{\underbrace{\text{goats like bunnies.}}_{\text{Sentence}}\ \text{and}\ \underbrace{\text{bunnies like astrophysics.}}_{\text{Sentence}}\ .}_{\text{Sentence}}$$

Sentences with periods

PunctuatedSentence → Sentence .

Sentence → Sentence and Sentence

Sentence → Sentence or Sentence

Sentence → Noun Verb Noun

Noun → goats

Noun → astrophysics

Noun → bunnies

Verb → like

Verb → see

- New rule adds a period only at end of sentence.
- Tokens are the 7 words plus the period (.)
- Grammar is ambiguous:

  goats like bunnies

  and bunnies like goats

  or bunnies like astrophysics

Grammars for programming languages

Grammar describes every possible legal expression

You could use the grammar for Java to list every possible Java program. (It would take forever.)

Grammar tells the Java compiler how to "parse" a Java program and defines what is **syntactically** legal (the compiler accepts)

docs.oracle.com/javase/specs/jls/se8/html/jls-2.html#jls-2.3

docs.oracle.com/javase/specs/jls/se8/html/jls-19.html

Grammar for simple expressions (not the best)

$E \rightarrow$ integer

$E \rightarrow (E + E)$

Simple expressions:

- An E can be an integer.
- An E can be '(' followed by an E followed by '+' followed by an E followed by ')'

Set of expressions defined by this grammar is a recursively-defined set

- Is language finite or infinite?
- Do recursive grammars always yield infinite languages?

Some legal expressions:

- 2
- (3 + 34)
- ((4+23) + 89)

Some illegal expressions:

- (3
- 3 + 4

Tokens of this grammar:

(+) and any integer

Parsing

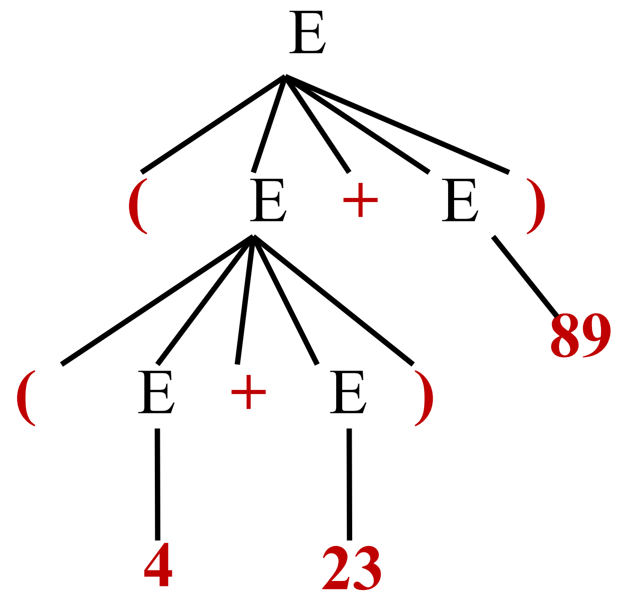
$E \rightarrow \text{integer}$
$E \rightarrow (E + E)$

Use a grammar in two ways:

- A grammar defines a *language* (i.e. the set of properly structured *sentences*)
- A grammar can be used to *parse* a *sentence* (thus, checking if a string is a *sentence* is in the *language*)

To *parse* a sentence is to build a *parse tree*: much like diagramming a sentence

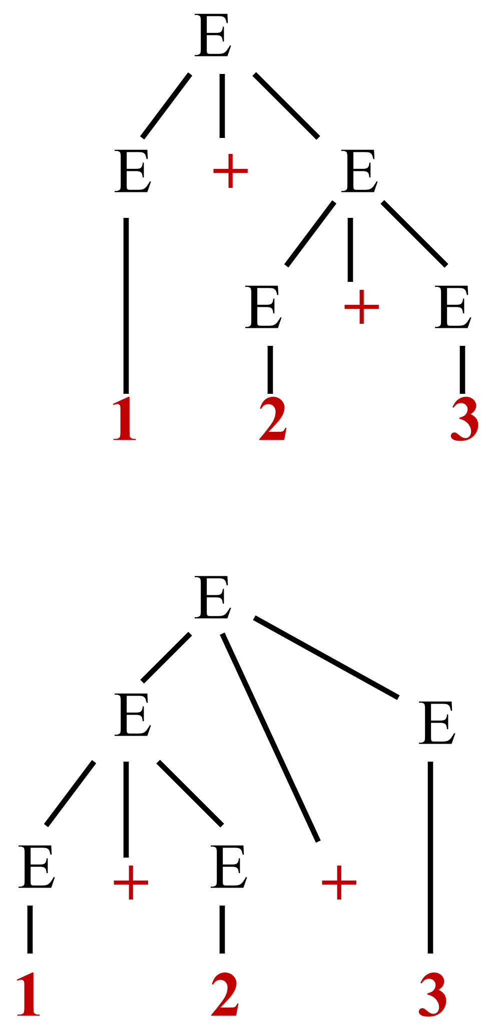
- Example: Show that
  $((4+23) + 89)$
  is a valid expression E by building a *parse tree*



Ambiguity

Grammar is ambiguous if it allows two parse trees for a sentence. The grammar below, using no parentheses, is ambiguous. The two parse trees to right show this. We don't know which + to evaluate first in the expression 1 + 2 + 3

E → integer
E → E + E



Recursive descent parsing

Write a set of mutually *recursive methods* to check if a sentence is in the language (show how to generate parse tree later).

One method for each nonterminal of the grammar. The method is completely determined by the rules for that nonterminal. On the next pages, we give a high-level version of the method for nonterminal E:

$$E \rightarrow \text{integer}$$

$$E \rightarrow (\ E + E\ )$$

Parsing an E

E → integer
E → (E + E)

/** Unprocessed input starts an E. Recognize that E, throwing away each piece from the input as it is recognized.
Return false if error is detected and true if no errors.
Upon return, processed tokens have been removed from input. */

public boolean parseE()

before call: already processed unprocessed

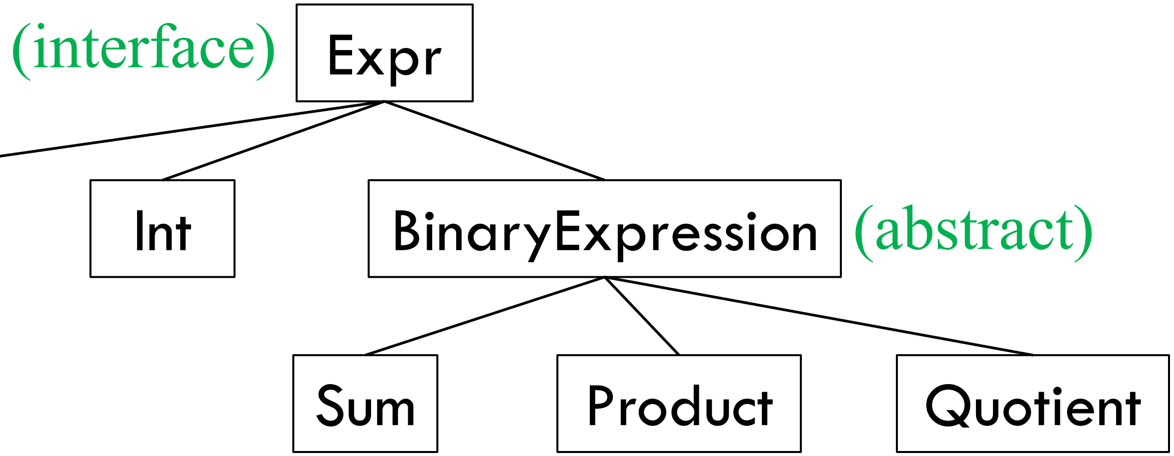
(2 + (4 + 8)) + 9)

after call:
(call returns true)

already processed unprocessed

(2 + (4 + 8)) + 9)

Expression trees: Class Hierarchy




```
public interface Expr {
    String infix(); // returns an infix representation
    int eval(); // returns the value of the expression
    // could easily also include prefix, postfix
}
```

Specification: /** Unprocessed input starts an E. …*/

$E \to$ integer
$E \to (E + E)$

public boolean parseE() {

if (first token is an integer) remove it from input and return true;

if (first token is not ‘(‘) return false **else** remove it from input;

if (!parseE()) return false;

if (first token is not ‘+‘) return false **else** remove it from input;

if (!parseE()) return false;

if (first token is not ‘)‘) return false **else** remove it from input;

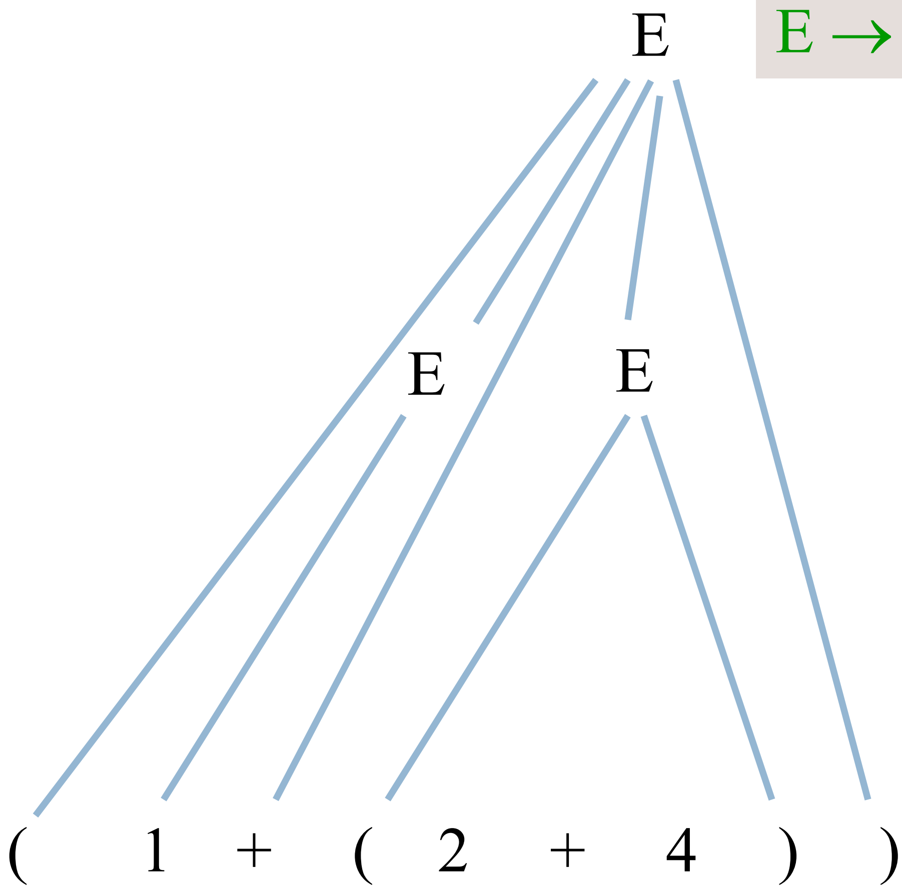
return true;

}

Illustration of parsing to check syntax

$E \rightarrow$ integer

$E \rightarrow (E + E)$



The scanner constructs tokens

An object scanner of class Scanner is in charge of the input String. It constructs the tokens from the String as necessary.

e.g. from the string “1464+634” build the token “1464”, the token “+”, and the token “634”.

It is ready to work with the part of the input string that has not yet been processed and has thrown away the part that is already processed, in left-to-right fashion.

$$\underbrace{(\ 2\ +\ (\ 4\ +\ 8\ )}_{\text{already processed}}\ \underbrace{+\ 9\ )}_{\text{unprocessed}}$$

Change parser to generate a tree

/** … Return an Expr for an E , or null if the string is illegal */


```
public Expr parseE() {
    if (next token is integer) {
        int val= the value of the token;
        remove the token from the input;
        return new Int(val);
    }
    if (next token is '(') remove it; else return null;
    Expr e1 = parseE();
    if (next token is '+') remove it; else return null;
    Expr e2 = parseE();
    if (next token is ')') remove it; else return null;
    return new Sum(e1, e2);
}
```

$E \rightarrow$ integer

$E \rightarrow (E + E)$

Grammar that gives precedence to * over +

E -> T { + T }

T -> F { * F }

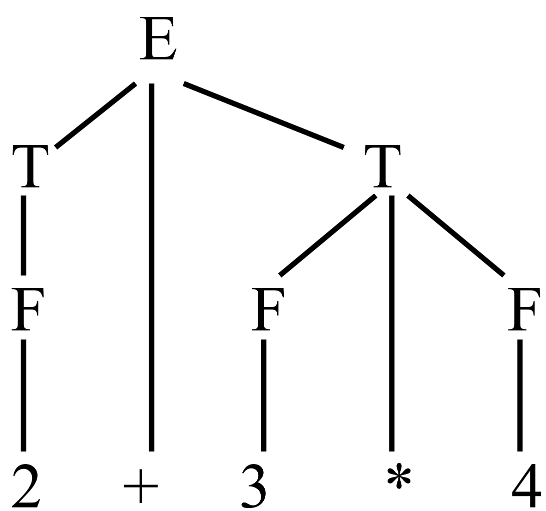
F -> integer

F -> (E)

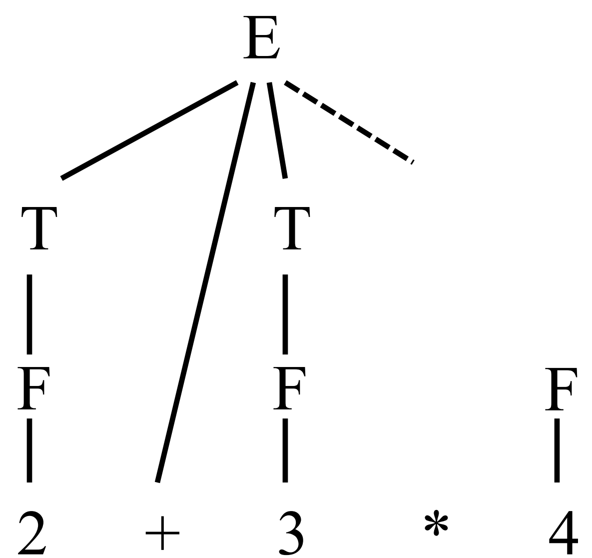
Notation: { xxx } means 0 or more occurrences of xxx.

E: Expression **T:** Term

F: Factor



says do * first



Try to do + first, can't complete tree

Does recursive descent always work?

Some grammars cannot be used for recursive descent

Trivial example (causes infinite recursion):

$S \rightarrow b$

$S \rightarrow Sa$

Can rewrite grammar

$S \rightarrow b$

$S \rightarrow bA$

$A \rightarrow a$

$A \rightarrow aA$

For some constructs, recursive descent is hard to use

Other parsing techniques exist – take the compiler writing course