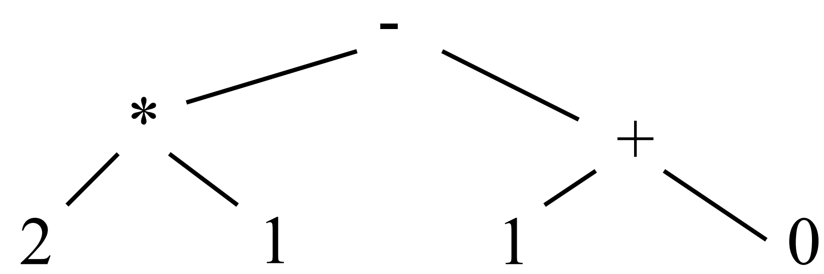
ASTS, GRAMMARS, PARSING, TREE TRAVERSALS

Lecture 13

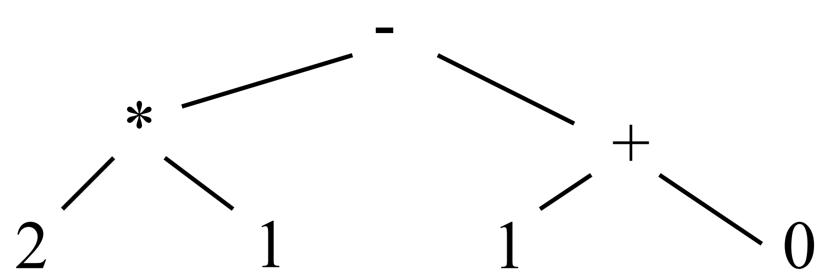
CS2110 – Fall 2017

Expression Trees

From last time: we can draw a **syntax tree** for the Java expression 2 * 1 – (1 + 0).



Pre-order, Post-order, and In-order

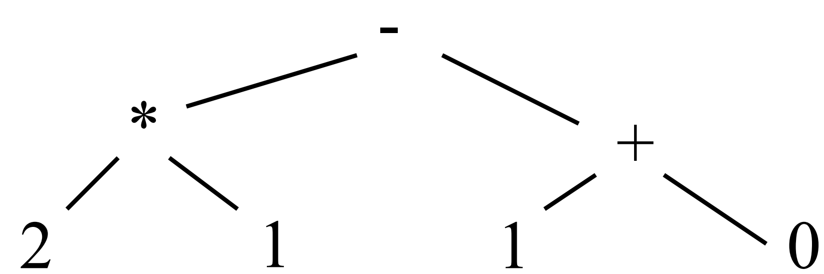


Pre-order traversal:

1. Visit the root
2. Visit the left subtree (in pre-order)
3. Visit the right subtree

- * 2 1 + 1 0

Pre-order, Post-order, and In-order

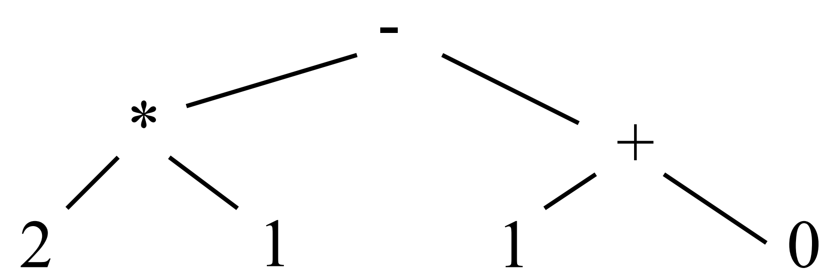


Pre-order traversal **- * 2 1 + 1 0**

Post-order traversal **2 1 * 1 0 + ***

1. Visit the left subtree (in post-order)
2. Visit the right subtree
3. Visit the root

Pre-order, Post-order, and In-order



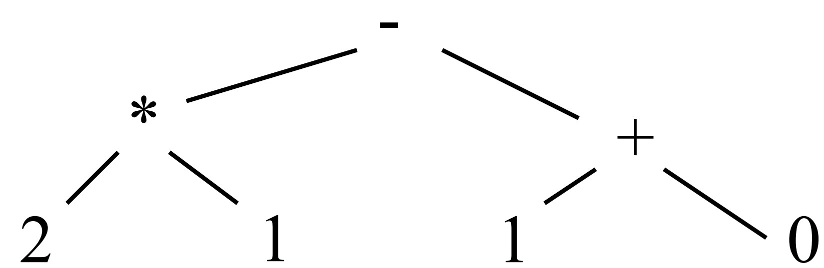
Pre-order traversal **- * 2 1 + 1 0**

Post-order traversal **2 1 * 1 0 + ***

In-order traversal **2 * 1 - 1 + 0**

1. Visit the left subtree (in-order)
2. Visit the root
3. Visit the right subtree

Pre-order, Post-order, and In-order



Pre-order traversal **- * 2 1 + 1 0**

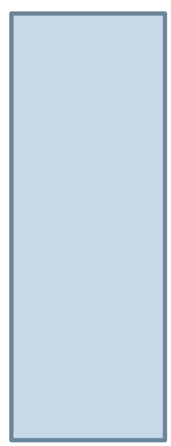
Post-order traversal **2 1 * 1 0 + ***

In-order traversal **(2 * 1) - (1 + 0)**

To avoid ambiguity, add parentheses around subtrees that contain operators.

In Defense of Postfix Notation

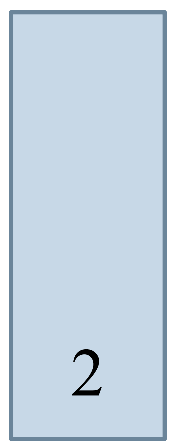
- Execute expressions in postfix notation by reading from left to right.
- Numbers: push onto the stack.
- Operators: pop the operands off the stack, do the operation, and push the result onto the stack.



2 1 * 1 0 + *

In Defense of Postfix Notation

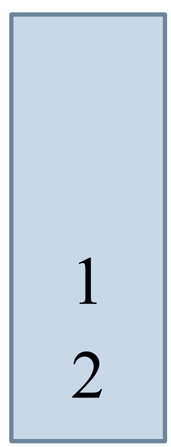
- Execute expressions in postfix notation by reading from left to right.
- Numbers: push onto the stack.
- Operators: pop the operands off the stack, do the operation, and push the result onto the stack.



1 * 1 0 + *

In Defense of Postfix Notation

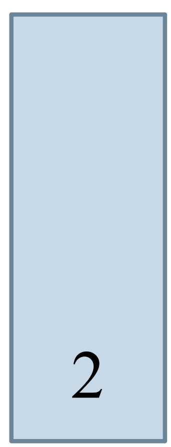
- Execute expressions in postfix notation by reading from left to right.
- Numbers: push onto the stack.
- Operators: pop the operands off the stack, do the operation, and push the result onto the stack.



* 1 0 + *

In Defense of Postfix Notation

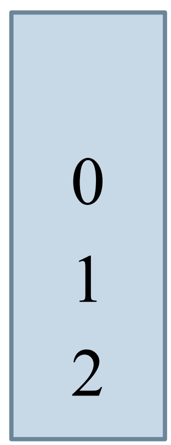
- Execute expressions in postfix notation by reading from left to right.
- Numbers: push onto the stack.
- Operators: pop the operands off the stack, do the operation, and push the result onto the stack.



1 0 + *

In Defense of Postfix Notation

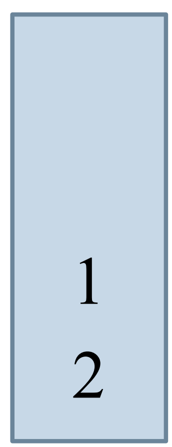
- Execute expressions in postfix notation by reading from left to right.
- Numbers: push onto the stack.
- Operators: pop the operands off the stack, do the operation, and push the result onto the stack.



+ *

In Defense of Postfix Notation

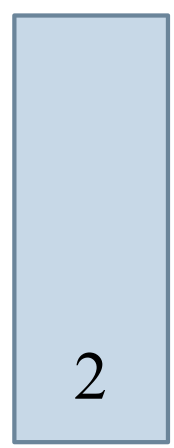
- Execute expressions in postfix notation by reading from left to right.
- Numbers: push onto the stack.
- Operators: pop the operands off the stack, do the operation, and push the result onto the stack.



*

In Defense of Postfix Notation

- Execute expressions in postfix notation by reading from left to right.
- Numbers: push onto the stack.
- Operators: pop the operands off the stack, do the operation, and push the result onto the stack.



In Defense of Postfix Notation

- Execute expressions in postfix notation by reading from left to right.
- Numbers: push onto the stack.
- Operators: pop the operands off the stack, do the operation, and push the result onto the stack.

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

a.k.a. “reverse Polish notation”

In Defense of Prefix Notation

- Function calls in most programming languages use prefix notation: like add(37, 5).
- Some languages (Lisp, Scheme, Racket) use prefix notation for *everything* to make the syntax simpler.

```
(define (fib n)
  (if (<= n 2)
      1
      (+ (fib (- n 1) (fib (- n 2)))))
```

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() + ")";
  }
}
```

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 sentences:
  The ate cat rat the
- How many legal sentences are there?
- How many legal Java programs are there?
- How can we check whether a string is a Java program?

Grammars

A **grammar** is a set of rules for generating the valid strings of a language.

Sentence	$\rightarrow$	Noun Verb Noun
Noun	$\rightarrow$	goats
Noun	$\rightarrow$	astrophysics
Noun	$\rightarrow$	bunnies
Verb	$\rightarrow$	like
Verb	$\rightarrow$	see

Grammars

A **grammar** is a set of rules for generating the valid strings of a language.

Sentence $\rightarrow$ Noun Verb Noun
Noun $\rightarrow$ goats
Noun $\rightarrow$ astrophysics
Noun $\rightarrow$ bunnies
Verb $\rightarrow$ like
Verb $\rightarrow$ see

Sentence

Grammars

A **grammar** is a set of rules for generating the valid strings of a language.

Sentence	$\rightarrow$	Noun Verb Noun
Noun	$\rightarrow$	goats
Noun	$\rightarrow$	astrophysics
Noun	$\rightarrow$	bunnies
Verb	$\rightarrow$	like
Verb	$\rightarrow$	see

Noun Verb Noun

Grammars

A **grammar** is a set of rules for generating the valid strings of a language.

Sentence	$\rightarrow$	Noun Verb Noun
Noun	$\rightarrow$	goats
Noun	$\rightarrow$	astrophysics
Noun	$\rightarrow$	bunnies
Verb	$\rightarrow$	like
Verb	$\rightarrow$	see

bunnies Verb Noun

Grammars

A **grammar** is a set of rules for generating the valid strings of a language.

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

bunnies like Noun

Grammars

A **grammar** is a set of rules for generating the valid strings of a language.

Sentence $\rightarrow$ Noun Verb Noun

Noun $\rightarrow$ goats

Noun $\rightarrow$ astrophysics

Noun $\rightarrow$ bunnies

Verb $\rightarrow$ like

Verb $\rightarrow$ see

bunnies like astrophysics

A Grammar

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

There are exactly 18 valid Sentences according to this grammar.

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

Grammars

A **grammar** is a set of rules for generating the valid strings of a language.

Sentence	$\rightarrow$	Noun Verb Noun
Noun	$\rightarrow$	goats
Noun	$\rightarrow$	astrophysics
Noun	$\rightarrow$	bunnies
Verb	$\rightarrow$	like
Verb	$\rightarrow$	see

bunnies like astrophysics
goats see bunnies
… (18 sentences total)

- 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

bunnies like astrophysics
goats see bunnies
bunnies like goats and goats see bunnies
… (infinite possibilities!)

The recursive definition of Sentence makes this grammar infinite.

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

A grammar describes every possible legal program.

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

A grammar also describes how to “parse” legal programs.

The Java compiler uses a grammar to translate your text file into a syntax tree—and to decide whether a program is legal.

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

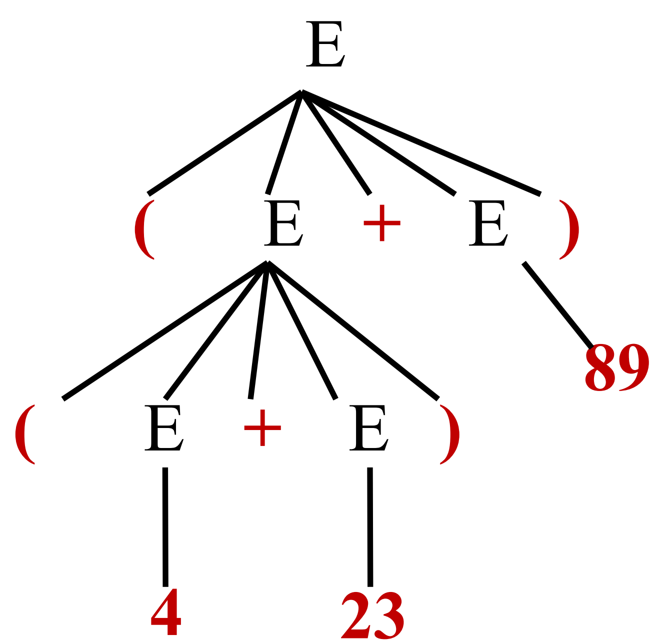
$E \rightarrow$ integer
$E \rightarrow (\ E + E\ )$

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

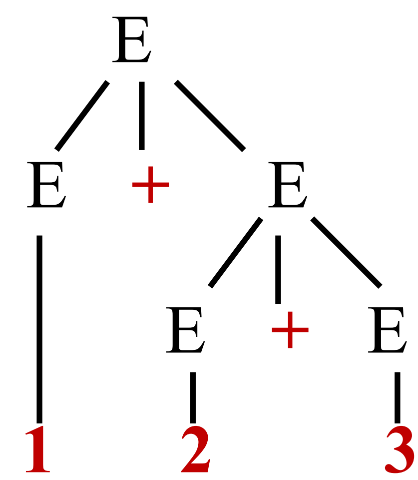
- Example: Show that
  ((4+23) + 89)
  is a valid expression E by building a *parse tree*

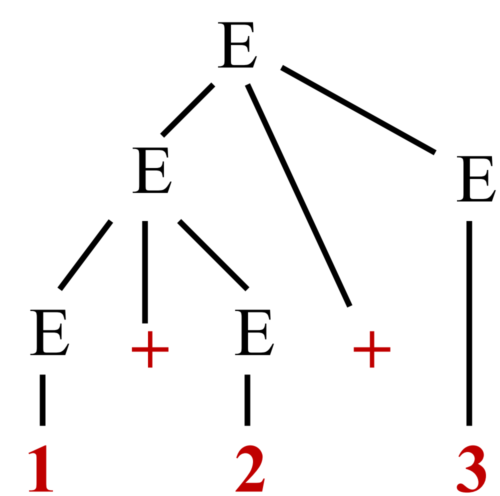


Ambiguity

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

$E \rightarrow$ integer
$E \rightarrow E + E$





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

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

/** 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()

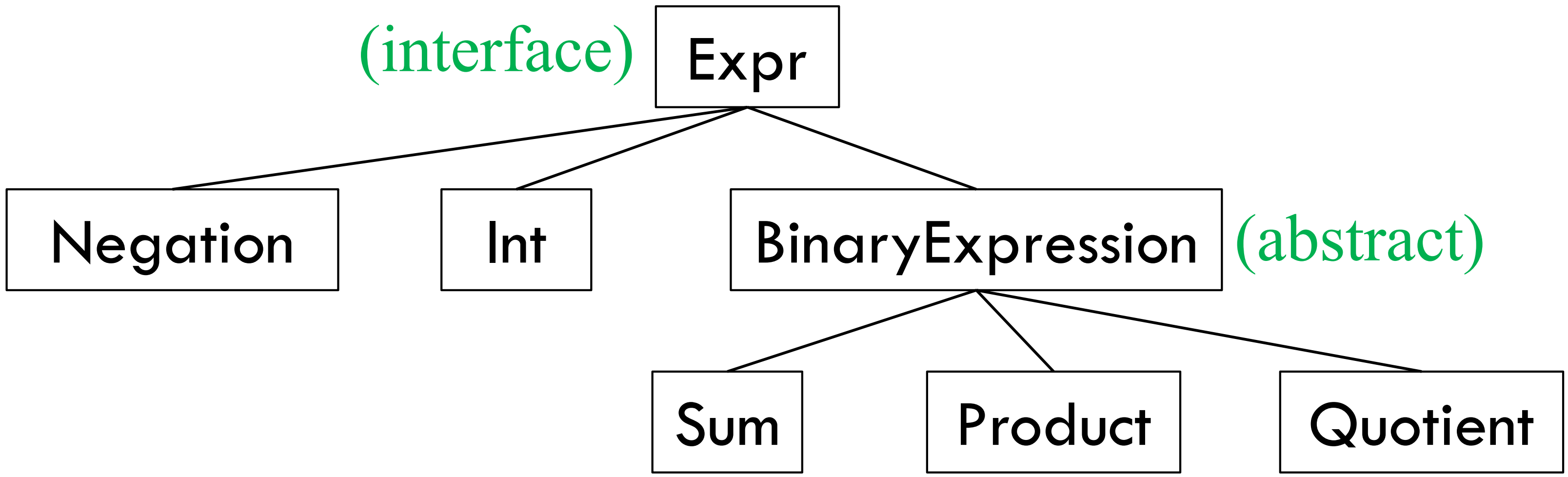
before call: already processed unprocessed

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

after call: already processed unprocessed
(call returns true)

(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. …*/

$E \rightarrow$ integer

$E \rightarrow (E + E)$

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;

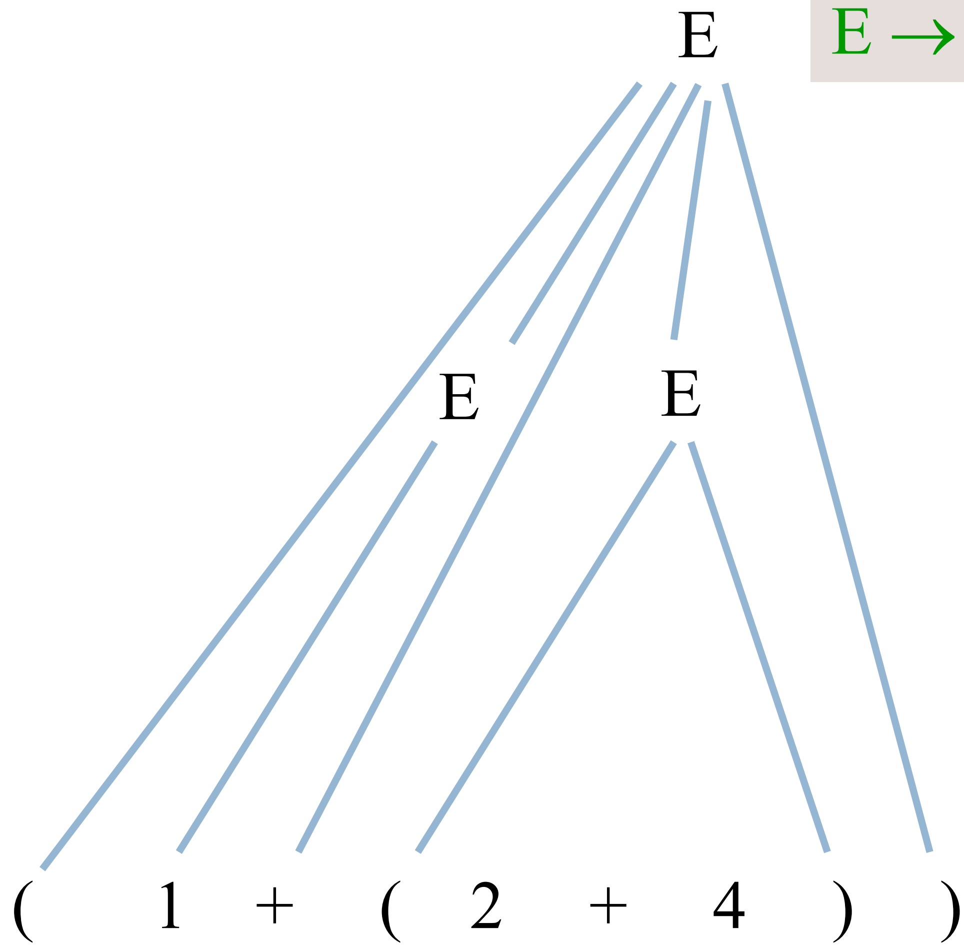
if (first token is not ‘)‘) return false **else** remove it from input;

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.

already processed unprocessed

(2 + (4 + 8) + 9)

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 → integer
E → (E + E)

Grammar that gives precedence to * over +

E -> T { + T }

T -> F { * F }

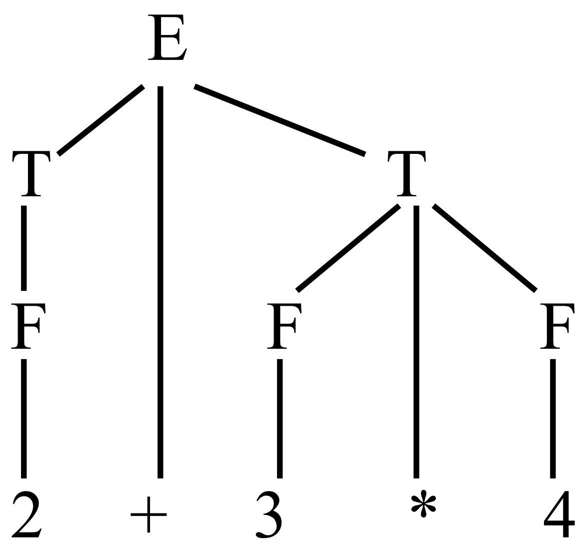
F -> integer

F -> (E)

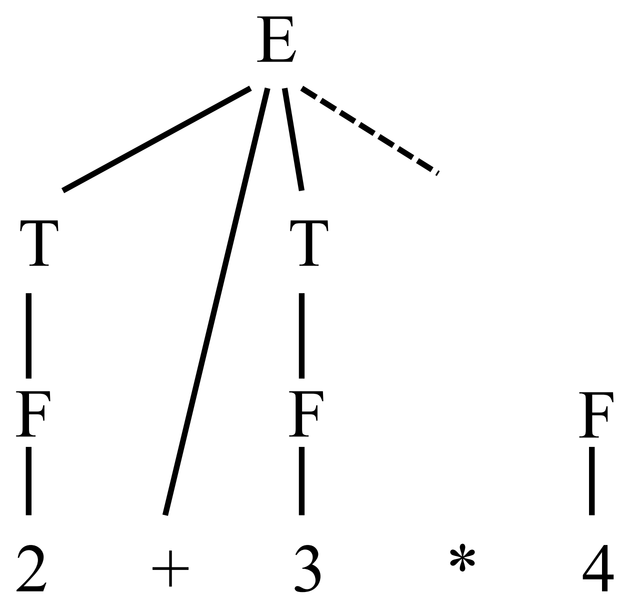
Notation: { xxx } means 0 or more occurrences of xxx.

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