ASTS, GRAMMARS, PARSING, TREE TRAVERSALS

Lecture 14

CS2110 – Fall 2018

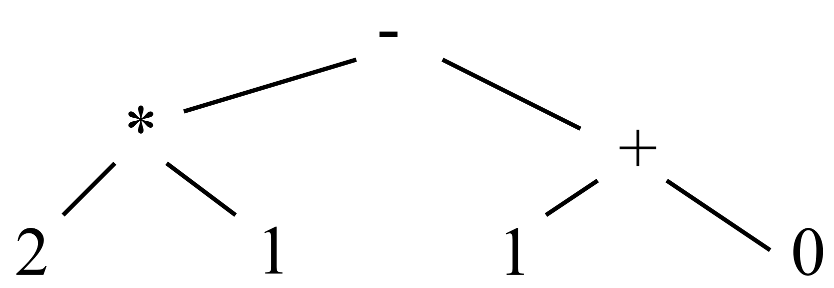
Announcements

- Today: The last day to request prelim regrades
- Assignment A4 due next Thursday night. Please work on it early and steadily. Watch the two videos on recursion on trees before working on A4!
- Next week's recitation. Learn about interfaces Iterator and Iterable. There will be 15 minutes of videos to watch. Then, in recitation, you will fix your A3 so that a foreach loop can be used on it.

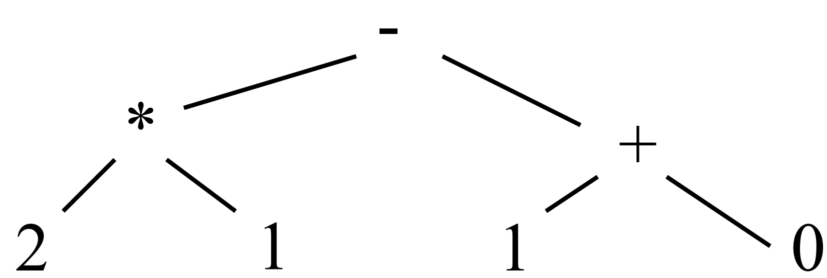
```
DLL<Integer> d= new DLL<Integer>();
 …
for (Integer i : d) { … }
```

Expression Trees

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



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

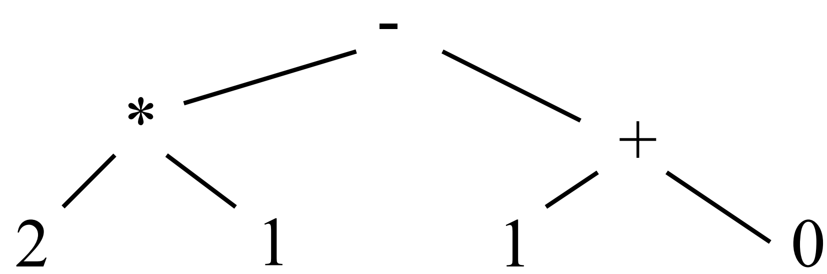


Pre-order traversal:

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

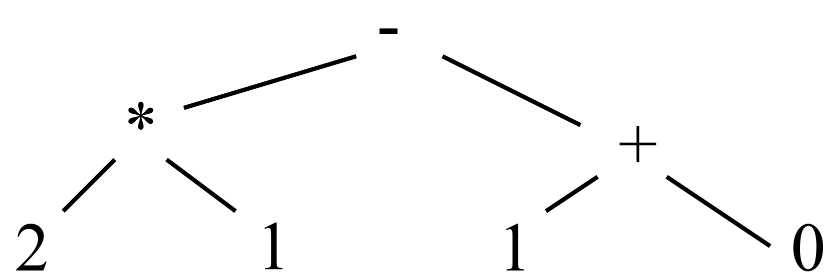


Pre-order traversal — - * 2 1 + 1 0

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



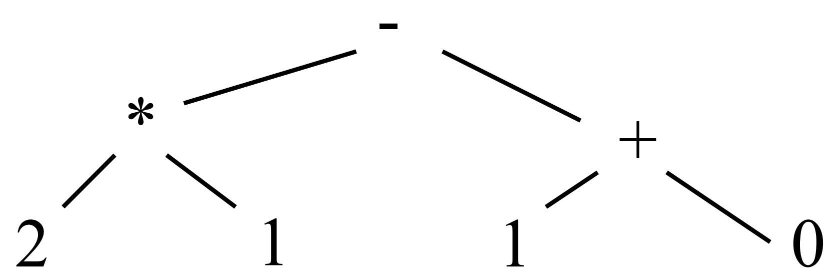
Pre-order traversal **- * 2 1 + 1 0**

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**

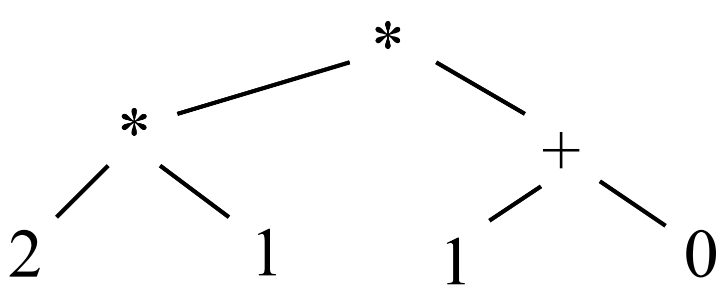
Post-order traversal **2 1 * 1 0 + -**

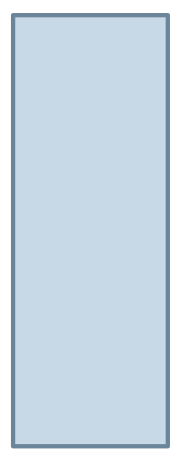
In-order traversal **(2 * 1) - (1 + 0)**

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.

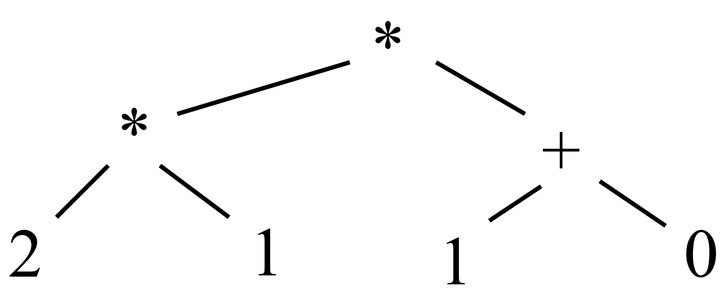


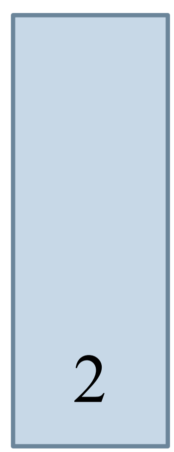


2 1 * 1 0 + *

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.

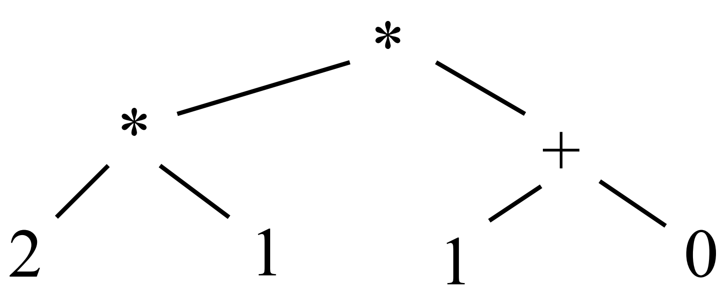


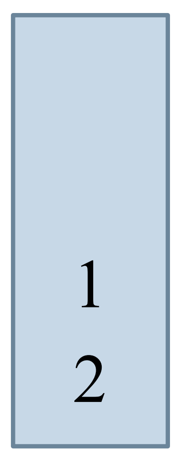


1 * 1 0 + *

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.

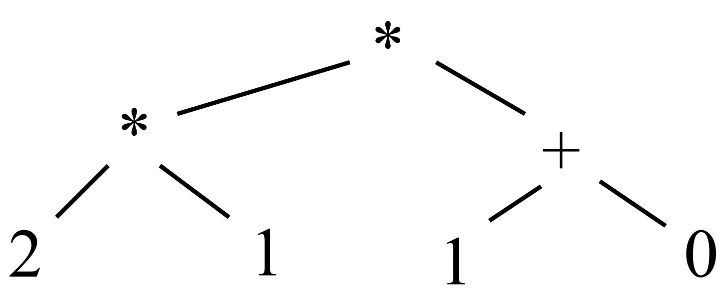


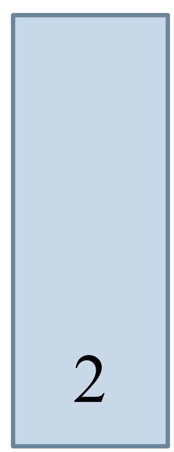


* 1 0 + *

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.

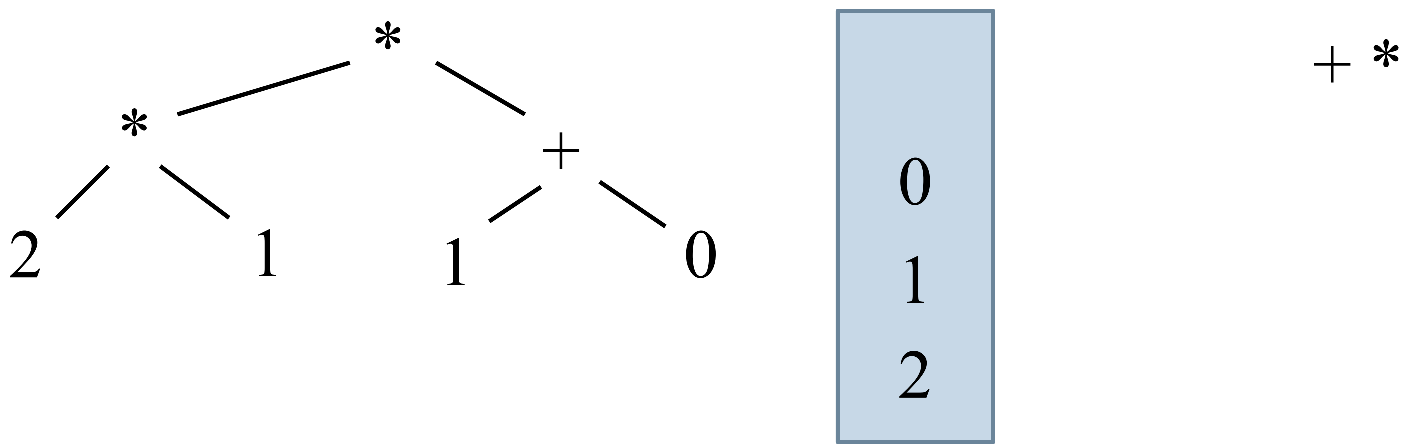




1 0 + *

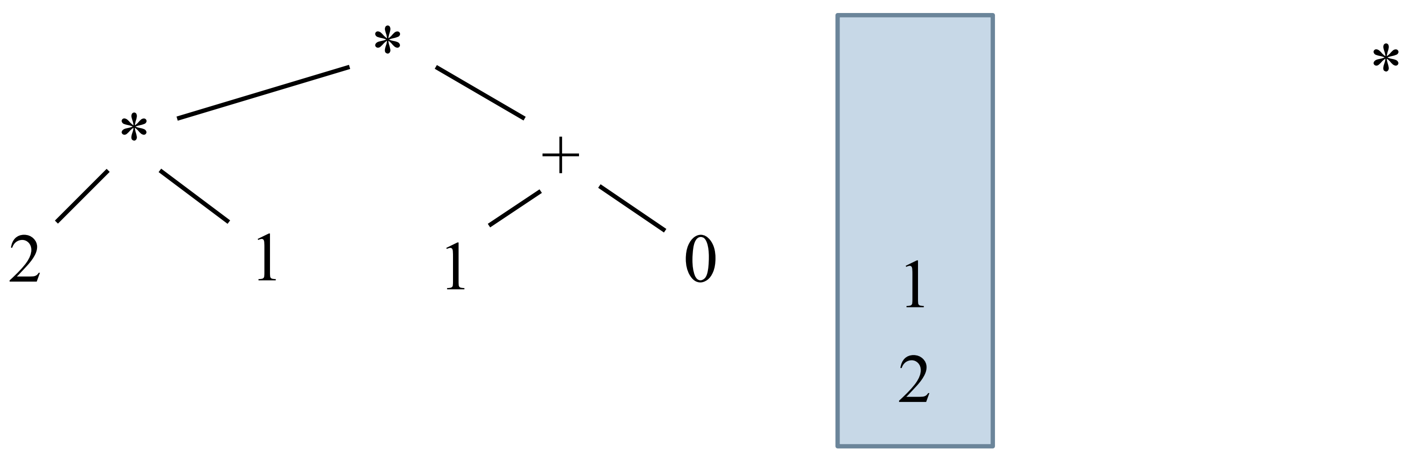
In Defense of Postfix Notation

- 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.



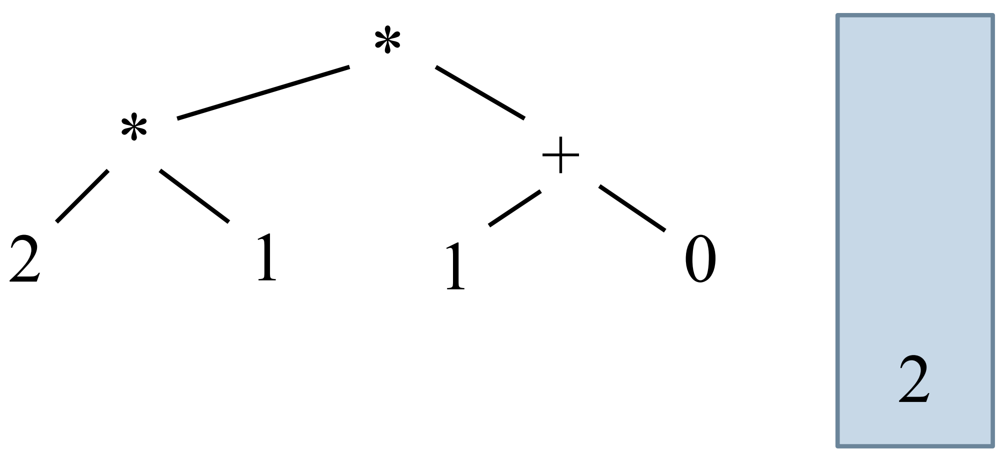
In Defense of Postfix Notation

- 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)))))
```

Determine tree from preorder and postorder

Suppose inorder is B C A E D

preorder is A B C D E

Can we determine the tree uniquely?

Determine tree from preorder and postorder

Suppose inorder is B C A E D

preorder is A B C D E

Can we determine the tree uniquely?

What is the root? preorder tells us: A

What comes before/after root A? Inorder tells us:

Before : B C

After: E D

Determine tree from preorder and postorder

Suppose inorder is B C A E D

preorder is A B C D E

The root is A.

Left subtree contains B C   Right subtree contains E D

Now figure out left, right subtrees *using the same method*.
From the above:

For left subtree
- inorder is: B C
- preorder is: B C
- root is: B
- Right subtree: C

For right subtree:
- inorder is: E D
- preorder is: D E
- root is: D
- left subtree: E

Expression trees: in code


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

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

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

Read → as "may be composed of"

Sentence → Noun Verb Noun

Noun → goats

Noun → astrophysics

Noun → bunnies

Verb → like

Verb → see

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

Sentence

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

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

bunnies like astrophysics

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.

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