Prelim 1

- Where: Kennedy Auditorium
- When:
  - A-Lib: 5:30-7
  - Lie-Z: 7:30-9
  - (unless we explicitly notified you otherwise)
Pointers to material

- 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-18.html
- Tree traversals – preorder, inorder, postorder: text, sections 23.13..23.15.
Expression trees

Can draw a tree for $2 \times 3 - (1 + 2 \times 4)$

```
public abstract class Exp {
    /* return the value of this Exp */
    public abstract int eval();
}
```
public abstract class Exp {
    /* return the value of this Exp */
    public abstract int eval();
}

public class Int extends Exp {
    int v;
    public int eval() {
        return v;
    }
}

public class Add extends Exp {
    Exp left;
    Exp right;
    public int eval() {
        return left.eval() + right.eval();
    }
}

Expression trees

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

*  + 2 3  + 1 4

prefix and postfix notation proposed by Jan Lukasiewicz in 1951

Postfix (we see it later) is often called RPN for Reverse Polish Notation
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.

Postfix notation

2 3 + 1 4 + *

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

Tree for \((2 + 3) \times (1 + 4)\)
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 \((2 + 3) \times (1 + 4)\)

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

\[(2 + 3) \times (1 + 4)\]
Expression trees

```java
public class Add extends Exp {
    Exp left;
    Exp right;
    /** Return the value of this exp. */
    public int eval() {return left.eval() + right.eval();}

    /** Return the preorder. */
    public String pre() {return "+  " + left.pre() + right.pre();}

    /** Return the postorder. */
    public String post() {return left.post() + right.post() + "+  " ;}
}

public abstract class Exp {
    public abstract int eval();
    public abstract String pre();
    public abstract String post();
}
```
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 $\rightarrow$ Noun Verb Noun

Noun $\rightarrow$ boys

Noun $\rightarrow$ girls

Noun $\rightarrow$ bunnies

Verb $\rightarrow$ 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 boys or girls or bunnies

AVerb can be like or see
A Grammar


Examples of Sentence:
- girls see bunnies
- bunnies like boys

- The words boys, girls, bunnies, like, see are called tokens or terminals
- The words Sentence, Noun, Verb are called nonterminals
A recursive grammar

Sentence \rightarrow Sentence and Sentence
Sentence \rightarrow Sentence or Sentence
Sentence \rightarrow Noun Verb Noun
Noun \rightarrow boys
Noun \rightarrow girls
Noun \rightarrow bunnies
Verb \rightarrow 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
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:

- girls like boys . and boys like bunnies .

Sentence Sentence Sentence
Sentences with periods

PunctuatedSentence $\rightarrow$ Sentence .
Sentence $\rightarrow$ Sentence and Sentence
Sentence $\rightarrow$ Sentence or Sentence
Sentence $\rightarrow$ Noun Verb Noun
Noun $\rightarrow$ boys
Noun $\rightarrow$ girls
Noun $\rightarrow$ bunnies
Verb $\rightarrow$ like
Verb $\rightarrow$ see

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

docs.oracle.com/javase/specs/jls/se8/html/jls-2.html#jls-2.3
Grammar for simple expressions (not the best)

E → integer
E → ( 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
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*
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

\[
E \rightarrow \text{integer} \\
E \rightarrow E + E
\]
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

/** 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
( 2 + ( 4 + 8 ) + 9 )
(call returns true)
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
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.

(   2   +   (   4    +    8    )
  +    9   )
Change parser to generate a tree

/** … Return a Tree for the E if no error. 
   Return null if there was an error*/

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

    if (first token is an integer) {
        Tree t = new Tree(the integer);
        Remove token from input;
        return t;
    }

    ...

    ...
}
Change parser to generate a tree

/** ... Return a Tree for the E if no error.
  Return null if there was an error*/

public Tree parseE() {
    if (first token is an integer) … ;
    if (first token is not ‘(‘ ) return null else remove it from input;
    Tree t1= parse(E); if (t1 == null) return null;
    if (first token is not ‘+‘ ) return null else remove it from input;
    Tree t2= parse(E); if (t2 == null) return null;
    if (first token is not ‘)‘ ) return false else remove it from input;
    return new Tree(t1, ‘+’, t2);
}
Code for a stack machine

Code for $2 + (3 + 4)$

- PUSH 2
- PUSH 3
- PUSH 4
- ADD
- ADD

ADD: remove two top values from stack, add them, and place result on stack

It’s postfix notation! $2 \ 3 \ 4 \ + \ +$

Stack

2
3
4

It’s postfix notation! $2 \ 3 \ 4 \ + \ +$
Code for a stack machine

Code for $2 + (3 + 4)$

- PUSH 2
- PUSH 3
- PUSH 4
- ADD
- ADD

ADD: remove two top values from stack, add them, and place result on stack

It’s postfix notation! $2 3 4 + +$
Use parser to generate code for a stack machine

Code for $2 + (3 + 4)$

PUSH 2
PUSH 3
PUSH 4
ADD
ADD

ADD: remove two top values from stack, add them, and place result on stack

parseE can generate code as follows:

- For integer $i$, return string “PUSH ” + i + “\n”
- For $(E1 + E2)$, return a string containing
  - Code for E1
  - Code for E2
  - “ADD\n”

It’s postfix notation! $2 \ 3 \ 4 \ + \ +$
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

E
 / \
T       T
 /     /
F       F
|     |
2     F
|   *
| 4

says do * first

E
 / \
T       T
 /     /
F       F
|     |
2     F
|   *
| 4

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):**

\[
\begin{align*}
S & \rightarrow b \\
S & \rightarrow Sa
\end{align*}
\]

Can rewrite grammar

\[
\begin{align*}
S & \rightarrow b \\
S & \rightarrow bA \\
A & \rightarrow a \\
A & \rightarrow aA
\end{align*}
\]

For some constructs, recursive descent is hard to use

Other parsing techniques exist – take the compiler writing course