Today: Parse Trees, Parsing, and Grammars

- Parse trees: text, section 23.36
- 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 
\[ 2 \times 3 - (1 + 2 \times 4) \]

```
interface Expr {
    /* evaluate 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
- 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 \((2 + 3) \times (1 + 4)\)
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!

<table>
<thead>
<tr>
<th>Infix</th>
<th>Prefix</th>
<th>Postfix</th>
</tr>
</thead>
<tbody>
<tr>
<td>(5 + 3) * 4</td>
<td>*(5 3 +) 4</td>
<td>5 3 * 4 +</td>
</tr>
<tr>
<td>5 + (3 * 4)</td>
<td>+5 3 4</td>
<td>5 3 * 4 +</td>
</tr>
<tr>
<td>1+2+3*4-7</td>
<td>+1 2 - 3 4 7</td>
<td>1 2 + 3 4 * 7 -</td>
</tr>
</tbody>
</table>

Expression trees: in code

```java
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

- Not all sequences of words are legal sentences
- 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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sentence</td>
<td>→ Noun Verb Noun</td>
</tr>
<tr>
<td>Noun</td>
<td>→ goats</td>
</tr>
<tr>
<td>Noun</td>
<td>→ astrophysics</td>
</tr>
<tr>
<td>Noun</td>
<td>→ bunnies</td>
</tr>
<tr>
<td>Verb</td>
<td>→ like</td>
</tr>
<tr>
<td></td>
<td>→ see</td>
</tr>
</tbody>
</table>

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 recursive grammar

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sentence</td>
<td>→ Sentence &amp; Sentence</td>
</tr>
<tr>
<td>Sentence</td>
<td>→ Sentence or Sentence</td>
</tr>
<tr>
<td>Sentence</td>
<td>→ Noun Verb Noun</td>
</tr>
<tr>
<td>Noun</td>
<td>→ goats</td>
</tr>
<tr>
<td>Noun</td>
<td>→ astrophysics</td>
</tr>
<tr>
<td>Noun</td>
<td>→ bunnies</td>
</tr>
<tr>
<td>Verb</td>
<td>→ like</td>
</tr>
<tr>
<td></td>
<td>→ see</td>
</tr>
</tbody>
</table>

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

Sentences with periods

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>PunctuatedSentence</td>
<td>→ Sentence .</td>
</tr>
<tr>
<td>Sentence</td>
<td>→ Sentence &amp; Sentence</td>
</tr>
<tr>
<td>Sentence</td>
<td>→ Sentence or Sentence</td>
</tr>
<tr>
<td>Sentence</td>
<td>→ Noun Verb Noun</td>
</tr>
<tr>
<td>Noun</td>
<td>→ goats</td>
</tr>
<tr>
<td>Noun</td>
<td>→ astrophysics</td>
</tr>
<tr>
<td>Noun</td>
<td>→ bunnies</td>
</tr>
<tr>
<td>Verb</td>
<td>→ like</td>
</tr>
<tr>
<td></td>
<td>→ see</td>
</tr>
</tbody>
</table>

- New rule adds a period only at end of sentence.
- Tokens are the 7 words plus the period (.)
- Grammar is ambiguous:
  - goats like bunnies
  - bunnies like goats
  - 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
- docs.oracle.com/javase/specs/jls/se8/html/jls-2.3.html
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

Parsing

E → integer
E → ( 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)

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

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 → integer
E → ( 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

(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

Some legal expressions:
- 2
- (3 + 34)
- ((4+23) + 89)

Some illegal expressions:
- (3
- 3 + 4

Tokens of this grammar: ( + ) and any integer

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 boolean parseE()

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

// could easily also include prefix, postfix

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 → integer
E → ( 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

(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

Recursive boolean parseE()
### Specification

/** Unprocessed input starts an E. ...*/

```java
class Expression {
    // Implementation...
}
```

```java
public boolean parseE() {
    if (first token is an integer) remove it from input and return true;
    if (!first token is not '(') return false;
    remove it from input;
    if (!parseE()) return false;
    if (!first token is not '+') return false;
    remove it from input;
    if (!parseE()) return false;
    if (!first token is not ')') return false;
    remove it from input;
    return true;
}
```

### Illustration of parsing to check syntax

```
E ® integer
E ® ( E + E )
```

Change parser to generate a tree

```java
/** ... 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 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);
}
```

### 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 )
```

### Grammar that gives precedence to * over +

```
E -> T { + T }  // says do * first
T -> F { * F }  // Try to do + first, can’t complete tree
F -> integer
F -> ( E )
```

### Does recursive descent always work?

Some grammars cannot be used for recursive descent

**Trivial example (causes infinite recursion):**

```
S -> b
S -> Sa
```

For some constructs, recursive descent is hard to use

Can rewrite grammar

```
S -> b  // Other parsing techniques exist – take the compiler writing course
S -> bA
A -> a
A -> aA
```