ADTS, GRAMMARS, PARSING, TREE TRAVERSALS

Lecture 13
CS2110 – Spring 2016

Points 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 \(2 \times 3 - (1 + 2 \times 4)\)

2
3
+ 
1
2
4
*

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

Expression trees

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

2
3
+ 
1
2
4
*

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();
  }
}

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

Prefix (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.

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

To help out, put parens around expressions with operators

Expression trees

Motivation for grammars

A Grammar

Not all sequences of words are legal sentences
The cat ate the 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

public class Add extends Exp {
    public abstract int eval();
    public abstract String pre();
    public abstract String post();
}

public abstract class Exp {
    public abstract int eval();
    public abstract String pre();
    public abstract String post();
}

Sentence → Noun Verb Noun
Noun → boys
Noun → girls
Noun → bunnies
Verb → like

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
A Verb can be like or see
### A Grammar

<table>
<thead>
<tr>
<th>A Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sentence  → Noun Verb Noun</td>
</tr>
<tr>
<td>Noun     → boys</td>
</tr>
<tr>
<td>Noun     → girls</td>
</tr>
<tr>
<td>Noun     → bunnies</td>
</tr>
<tr>
<td>Verb     → like</td>
</tr>
<tr>
<td>Verb     → see</td>
</tr>
</tbody>
</table>

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

**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

<table>
<thead>
<tr>
<th>A recursive grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sentence  → Sentence and Sentence</td>
</tr>
<tr>
<td>Sentence  → Sentence or Sentence</td>
</tr>
<tr>
<td>Sentence  → Noun Verb Noun</td>
</tr>
<tr>
<td>Noun     → boys</td>
</tr>
<tr>
<td>Noun     → girls</td>
</tr>
<tr>
<td>Noun     → bunnies</td>
</tr>
<tr>
<td>Verb     → like</td>
</tr>
<tr>
<td>Verb     → 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?**
- Recursive definition of Sentence

### Detour

<table>
<thead>
<tr>
<th>Detour</th>
</tr>
</thead>
<tbody>
<tr>
<td>What if we want to add a period at the end of every sentence?</td>
</tr>
<tr>
<td>Sentence  → Sentence and Sentence .</td>
</tr>
<tr>
<td>Sentence  → Sentence or Sentence .</td>
</tr>
<tr>
<td>Sentence  → Noun Verb Noun .</td>
</tr>
<tr>
<td>Noun     → ...</td>
</tr>
</tbody>
</table>

**Does this work?**
- No! This produces sentences like:
  - girls like boys. and boys like bunnies...

### Sentences with periods

<table>
<thead>
<tr>
<th>Sentences with periods</th>
</tr>
</thead>
<tbody>
<tr>
<td>PunctuatedSentence  → Sentence .</td>
</tr>
<tr>
<td>Sentence  → Sentence and Sentence</td>
</tr>
<tr>
<td>Sentence  → Sentence or Sentence</td>
</tr>
<tr>
<td>Sentence  → Noun Verb Noun</td>
</tr>
<tr>
<td>Noun     → boys</td>
</tr>
<tr>
<td>Noun     → girls</td>
</tr>
<tr>
<td>Noun     → bunnies</td>
</tr>
<tr>
<td>Verb     → like</td>
</tr>
<tr>
<td>Verb     → 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:
  - boys like girls
  - and girls like boys
  - or girls like bunnies

### Grammars for programming languages

<table>
<thead>
<tr>
<th>Grammars for programming languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grammar describes every possible legal expression</td>
</tr>
<tr>
<td>You could use the grammar for Java to list every possible Java program. (It would take forever.)</td>
</tr>
</tbody>
</table>

**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](https://docs.oracle.com/javase/specs/jls/se8/html/jls-2.html#jls-2.3)

### Grammar for simple expressions (not the best)

<table>
<thead>
<tr>
<th>Grammar for simple expressions (not the best)</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → integer</td>
</tr>
<tr>
<td>E → ( E + E )</td>
</tr>
</tbody>
</table>

**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 + 4)
- ((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 \( 1 + 2 + 3 \).

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 \( E \):

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

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

Parsing an \( E \):

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

/** 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() {
    //...
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 “1 464+634” build the token “1 464”, 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.

<table>
<thead>
<tr>
<th>already processed</th>
<th>unprocessed</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2 + (4 + 8)) + 9</td>
<td></td>
</tr>
</tbody>
</table>

Change parser to generate a tree

```java
/** ... 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 not '(') return null else remove it from input;
    Tree t1 = parse(E);
    if (t1 == null) return null;
    if (first token is not '+') return false 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);
}
```

Use parser to generate code for a stack machine

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 + +
```

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 + +
```

parseE can generate code as follows:

- For integer i, return string “PUSH “i” “in””
- For (E1 + E2), return a string containing • Code for E1
  • Code for E2
  • “ADDn”
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

2 + 3 * 4

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 → b
S → Sa

Can rewrite grammar
S → b
S → bA
A → a
A → aA

For some constructs, recursive descent is hard to use

Other parsing techniques exist – take the compiler writing course