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 \(2 \times 1 - (1 + 0)\).

\[
\begin{array}{c}
2 \\
\times \\
1 \\
\times \\
1 \\
+ \\
0
\end{array}
\]

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 \(\times\) 1 \(-\) (1 \(\times\) 0)

**Pre-order traversal**
- \(\times\) 2 \(\times\) 1 \(+\) \(\times\) 0

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

2 \(\times\) 1 \(-\) (1 \(\times\) 0)

**Post-order traversal**
2 \(\times\) 1 \(-\) \(\times\) (1 \(\times\) 0)

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

2 \(\times\) 1 \(-\) (1 \(\times\) 0)

**In-order traversal**
(2 \(\times\) 1) \(-\) (1 \(\times\) 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 + *

1 * 1 0 + *

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 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)

(fib 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 {  // returns an infix representation
  String infix();
  int eval();  // returns the value of the expression
}
```

```java
public class Int implements Expr {  // returns an infix representation
  private int v;
  public int eval() { return v; }
  public String infix() { return "" + v + "";
  }
}
```

```java
public class Sum implements Expr {  // returns an infix representation
  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 sat in the hat ate the big rat on the mat slowly, then got sick.

- Not all sequences of words are sentences:
  - The cat ate the rat
  - How many legal sentences are there?
  - How many legal Java programs are there?
  - How can we check whether a string is a Java program?
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
A Grammar

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

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

There are exactly 18 valid Sentences according to this grammar.

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

• 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
Verb → see

bunnies like astrophysics
bunnies 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:
goats like bunnies. and bunnies like astrophysics.

The recursive definition of Sentence makes this grammar infinite.

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 → 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 )

Expression trees: Class Hierarchy

public interface Expr { }
**Unprocessed input starts an E. ...*/

```java
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;
}
```

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.

- Example: 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.

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

Grammar that gives precedence to * over +

- `E -> T { + T }` - says do * first
- `F -> ( E )` - Try to do + first, can’t complete tree
- `F -> T { * F }` - Notation: \{ xxx \} means 0 or more occurrences of xxx.

<table>
<thead>
<tr>
<th>E: Expression</th>
<th>T: Term</th>
<th>F: Factor</th>
</tr>
</thead>
</table>

Grammar:
- `E -> integer`
- `E -> ( E + E )`

Illustration of parsing to check syntax

Change parser to generate a tree

```java
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);
}
```

Change parser to generate a tree

```java
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);
}
```

Some grammars cannot be used for recursive descent

- Trivial example (causes infinite recursion):
  - `S -> b`
  - `S -> Sa`

Can rewrite grammar

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

Does recursive descent always work?

```java
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);
}
```