Grammars & Parsing

Lecture 9
CS2110 – Fall 2011
Java Tips

-Declare fields and methods
  public if they are to be visible outside the class; helper methods and private data should be declared private
-Constants that will never be changed should be declared final
-Public classes should appear in a file of the same name
-Two kinds of boolean operators:
  - e1 & e2: evaluate both and compute their conjunction
  - e1 && e2: evaluate e1; don’t evaluate e2 unless necessary

- instead of
  if (s.equals("")) {
    f = true;
  } else {
    f = false;
  }
  write
  f = s.equals("");

- instead of
  if (s.equals("")) {
    f = a;
  } else {
    f = b;
  }
  write
  f = s.equals("")? a : b;
Application of Recursion

• So far, we have discussed recursion on integers
  ▪ factorial, Fibonacci, combinations, $a^n$

• Now we’ll consider a new application that shows off the full power of recursion: parsing

• Parsing has numerous applications: compilers, data retrieval, data mining, …
Motivation

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.
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 programs are there?
- Are all Java programs that compile legal programs?
- How do we know what programs are legal?

A Grammar

Sentence ::= Noun Verb Noun
Noun ::= boys | girls | bunnies
Verb ::= like | 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 ‘boys’ or ‘girls’ or ‘bunnies’
  ▪ A Verb can be ‘like’ or ‘see’

• Examples of Sentence:
  ▪ boys see bunnies
  ▪ bunnies like girls
  ▪ ...

• Grammar: set of rules for generating sentences in a language
• White space between words does not matter
• The words boys, girls, bunnies, like, see are called tokens or terminals
• The words Sentence, Noun, Verb are called syntactic classes or nonterminals
• This is a very boring grammar because the set of Sentences is finite (exactly 18)
A Recursive Grammar

Sentence ::= Sentence and Sentence
| Sentence or Sentence
| Noun Verb Noun
Noun ::= boys | girls | bunnies
Verb ::= like | see

• This grammar is more interesting than the last one because the set of Sentences is infinite

• Examples of Sentences in this language:
  ▪ boys like girls
  ▪ boys like girls and girls like bunnies
  ▪ boys like girls and girls like bunnies and girls like bunnies
  ▪ boys like girls and girls like bunnies and girls like bunnies and girls like bunnies
  ▪ ...

• What makes this set infinite?
  Answer:
    ▪ Recursive definition of Sentence
Detour

• What if we want to add a period at the end of every sentence?

Sentence ::= Sentence and Sentence .
  | Sentence or 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

TopLevelSentence ::= Sentence . 
Sentence ::= Sentence and Sentence 
   | Sentence or Sentence 
   | Noun Verb Noun 
Noun ::= boys | girls | bunnies 
Verb ::= like | see 

- Add a new rule that adds a period only at the end of the sentence.

- The tokens here are the 7 words plus the period (.)

- This grammar is ambiguous:
  boys like girls and girls like boys or girls like bunnies
Grammar for Simple Expressions

\[
E ::= \text{integer | } (E + E)
\]

- 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 an inductively-defined set
  - Is the language finite or infinite?
  - Do recursive grammars always yield infinite languages?

- Here are some legal expressions:
  - 2
  - (3 + 34)
  - ((4+23) + 89)
  - ((89 + 23) + (23 + (34+12)))

- Here are some illegal expressions:
  - (3
  - 3 + 4

- The tokens in this grammar are (, +, ), and any integer
**Parsing**

- Grammars can be used 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 the *sentence* is in the *language*)

- To *parse* a sentence is to build a *parse tree*
  - This is much like *diagramming a sentence*

- Example: Show that 
\[ ((4+23) + 89) \]
is a valid expression \( E \) by building a *parse tree*
Recursive Descent Parsing

• Idea: Use the grammar to design a recursive program to check if a sentence is in the language
• To parse an expression E, for instance
  ▪ We look for each terminal (i.e., each token)
  ▪ Each nonterminal (e.g., E) can handle itself by using a recursive call
• The grammar tells how to write the program!

```java
boolean parseE( ) {
    if (first token is an integer) return true;
    if (first token is '(') {
        scan past '(' token;
        parseE( );
        scan past '+' token;
        parseE( );
        scan past ')') token;
        return true;
    }
    return false;
}
```
Java Code for Parsing E

```java
public static Node parseE(Scanner scanner) {
    if (scanner.hasNextInt()) {
        int data = scanner.nextInt();
        return new Node(data);
    }
    check(scanner, '(');
    left = parseE(scanner);
    check(scanner, '+');
    right = parseE(scanner);
    check(scanner, ')');
    return new Node(left, right, '+');
}
```
Responding to Invalid Input

- **Parsing does two things:**
  - checks for validity (Is the input a valid sentence?)
  - constructs the parse tree (usually called an AST or abstract syntax tree)

- **Q:** How should we respond to invalid input?

- **A:** Throw an exception with as much information for the user as possible
  - the nature of the error
  - approximately where in the input it occurred
Using a Parser to Generate Code

• We can modify the parser so that it generates stack code to evaluate arithmetic expressions:
  
  \[
  \begin{align*}
  &2 & \text{PUSH 2} & \text{STOP} \\
  &(2 + 3) & \text{PUSH 2} & \text{PUSH 3} & \text{ADD} & \text{STOP}
  \end{align*}
  \]

• Goal: Modify parseE to return a string containing stack code for expression it has parsed

• Method parseE can generate code in a recursive way:
  
  ▪ For integer \( i \), it returns string “PUSH ” + \( i \) “\n”
  
  ▪ For \((E1 + E2)\),
    
    ✗ Recursive calls for \( E1 \) and \( E2 \) return code strings \( c1 \) and \( c2 \), respectively
    
    ✗ Return \( c1 + c2 + “\text{ADD}\"\text{\n}”\)
  
  ▪ Top-level method appends a STOP command
Does Recursive Descent Always Work?

• No – some grammars cannot be used with recursive descent
  ▪ A trivial example (causes infinite recursion):
    \[ S ::= b \mid Sa \]

• Can rewrite grammar
  \[ S ::= b \mid bA \]
  \[ A ::= a \mid aA \]

• Sometimes recursive descent is hard to use
  ▪ There are more powerful parsing techniques (not covered in this course)

• Nowadays, there are automated parser and tokenizer generators
  ▪ you write down the grammar, it produces the parser and tokenizer automatically
Syntactic Ambiguity

• Sometimes a sentence has more than one parse tree
  
  S ::= A | axB
  A ::= x | aAb
  B ::= b | bB

  ▪ The string aaxbb can be parsed in two ways

• This kind of ambiguity sometimes shows up in programming languages

if E1 then if E2 then S1 else S2

Which then does the else go with?

• This ambiguity actually affects the program’s meaning

• How do we resolve this?
  ▪ Operator precedence – e.g., always parse 1 + 2 * 3 as 1 + (2 * 3), not (1 + 2) * 3
  ▪ Provide an extra non-grammar rule – e.g., the else goes with the closest if – similar to operator precedence
  ▪ Modify the language – e.g., an if-statement must end with a ‘fi’ – similar to requiring parentheses
  ▪ Other methods – e.g., Python uses depth of indentation
Exercises

Write a grammar and recursive-descent parser for

• palindromes:
  mom  dad  I prefer pi  race car
  A man, a plan, a canal: Panama
  murder for a jar of red rum  sex at noon taxes

• strings of the form  $A^nB^n$ for some $n \geq 0$:
  AB  AABB  AAAAAAABBBBBBB

• Java identifiers:
  a letter, followed by any number of letters or digits

• decimal integers:
  an optional minus sign (−) followed by one or more digits 0-9
package tableau.test;
import static org.junit.Assert.*;
import org.junit.Test;
// other imports

public class ParserTest {
    protected ASTNode parse(String formula) throws ParseException {
        return new Parser(formula).parseFormula();
    }

    @Test
    public void basicTest() throws ParseException {
        final Collection<String> formulas = ...;
        for (String formula : formulas) {
            assertEquals(formula, parse(formula).toString());
        }
    }

    // other tests
}
JUnit

```java
@Test
public void precedenceTest() throws ParseException {
    assertEquals(TokenType.NOT, parse("~P").getType());
    assertEquals(TokenType.AND, parse("P & Q").getType());
    assertEquals(TokenType.IMPLIES, parse("~P -> P").getType());
    assertEquals(TokenType.OR, parse("(P & Q) | P").getType());
    assertEquals(TokenType.IMPLIES, parse("Q -> P -> Q").getType());
    assertEquals(TokenType.IFF, parse("~(P & Q) <-> ~P | ~Q").getType());
}

@Test(expected = ParseException.class)
public void exceptionTest1() throws ParseException {
    parse("((sss -> ttt) -> uuu) qqq");
}
```
JUnit

Hospital h = new Hospital();

@Test
public void testHospital() {
    assertNotNull("First room is null", h.firstRoom);
    assertEquals("First room number is not 100",
                 100, h.firstRoom.roomNumber);
    assertNotNull("No patient in first room", h.firstRoom.patient);
    assertNotNull("No doctor", h.doctor);
    assertSame("Doctor is not initially in the first room",
               h.firstRoom, h.doctor.location);
}

@Test
public void testDone() {
    h.doctor.medicine = 1;
    assertFalse("Asserted done when not", h.xdone());
    h.doctor.medicine = 0;
    assertTrue("Did not detect exhausted medicine", h.xdone());
}