Grammars & Parsing

Lecture 4
CS211 – Fall 2006

Announcements

- Java Reminder
  - Any "significant" class should be declared *public* and should appear in a file whose name matches the class name

Application of Recursion

- So far, we have discussed recursion on integers
  - Factorial, fibonacci, combinations, a^n
- Let us now 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?

http://java.sun.com/docs/books/jls/third_edition/html/syntax.html

A Grammar

Sentence → Noun Verb Noun
Noun → boys
Noun → girls
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 'boys' or 'girls' or 'bunnies
  - A Verb can be 'like' or 'see'
- Grammar: set of rules for generating sentences in a language
- Examples of Sentence:
  - boys see bunnies
  - bunnies like girls
  - …
- Note: white space between words does not matter
  - The *tokens* here are words
  - This grammar has 5 *tokens*
- This is a very boring grammar because the set of Sentences is finite (exactly 18 sentences)

A Recursive Grammar

Sentence → Sentence and Sentence
Sentence → Sentence or Sentence
Sentence → Noun Verb Noun
Noun → boys
Noun → girls
Noun → bunnies
Verb → like
Verb → see

- This grammar is more interesting than the one in the last slide 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 → Sentence or Sentence .

Sentence → Noun Verb Noun .

Noun → ...

- Does this work?
- No! This produces sentences like:

girls like boys . and boys like bunnies . .

Sentences with Periods

TopLevelSentence → Sentence .
Sentence → Sentence and Sentence
Sentence → Sentence or Sentence
Sentence → Noun Verb Noun
Noun → boys
Noun → girls
Noun → bunnies
Verb → like
Verb → see

- Add a new rule that adds a period only at the end of the sentence.
- Thought exercise: How does this work?
- The tokens here are the 5 words plus the period (.)

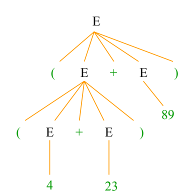
Grammar for Simple Expressions

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?
- 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*)
- One way 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!!

Pseudo Code:

```
public boolean parseE( ):
   if first token is an integer: return true;
   if first token is "(":
         parseE( );
         Make sure there is a "+" token;
         parseE( );
         Make sure there is a ")" token;
         return true;
   return false;
```

Java Code for Parsing E

```
public static boolean parseE (Scanner scanner) {
    if (scanner.hasNextInt()) {
        scanner.nextInt();
        return true;
    }
    return check(scanner, "(") &&
        parseE(scanner) &&
        check(scanner, "+") &&
        parseE(scanner) &&
        check(scanner, ")");
}
```

- Recall: Java supports two kinds of Boolean operators:
  - E1 & E2: evaluate both and compute their conjunction
  - E1 && E2: evaluate E1; don't evaluate E2 unless necessary

Detour: Java Exceptions

- Parsing is used in many ways; for example:
  - Code generation
  - Code interpretation
  - Building parse trees
- For all these examples, parsing does two things:
  - It returns useful data (e.g., code, an answer, a parse tree)
  - It checks for validity (i.e., is the input a valid *sentence*?)
- *Exceptions* allow us to respond to invalid input without complicating our code

Exceptions

- Exceptions are usually thrown to indicate that something bad has happened
  - IOException on failure to open or read a file
  - ClassCastException if attempted to cast an object to a type that is not a supertype of the dynamic type of the object
  - NullPointerException if tried to dereference null
  - ArrayIndexOutOfBoundsException if tried to access an array element at index $i < 0$ or $\geq$ the length of the array
- In our case (parsing), we need to indicate invalid syntax

Handling Exceptions

- Exceptions can be caught by the program using a try/catch block
- catch clauses are called *exception handlers*

```
Integer x = null;
try {
   x = (Integer)y;
   System.out.println(x.intValue());
} catch (ClassCastException e) {
   System.out.println("y was not an Integer");
} catch (NullPointerException e) {
   System.out.println("y was null");
}
```

Defining Your Own Exceptions

- An exception is an object (like everything else in Java)
  - You can define your own exceptions and throw them

```
class MyOwnException extends Exception {}

...

if (input == null) {
   throw new MyOwnException();
}
```

The throws Clause

- In general, any exception you throw must be either *declared* in the method header or *caught*

```
void foo(int input) throws MyOwnException {
  if (input == null) {
    throw new MyOwnException();
  }
  ...
}
```

- Note: throws means "can throw", not "does throw"
- Subtypes of RuntimeException do not have to be declared (e.g., NullPointerException, ClassCastException)
  - These represent exceptions that can occur during "normal operation of the Java Virtual Machine"

How Exceptions are Handled

- If the exception is thrown from *inside* a try/catch block with a handler for that exception (or a superclass of the exception), then that handler is executed
  - Otherwise, the method terminates abruptly and control is passed back to the calling method
- If the calling method can handle the exception (i.e., if the call occurred within a try/catch block with a handler for that exception) then that handler is executed
  - Otherwise, the calling method terminates abruptly, etc.
- If *none* of the calling methods handle the exception, the entire program terminates with an error message

Using a Parser to Generate Code

- We can modify the parser so that it generates stack code to evaluate arithmetic expressions:

2	PUSH 2 STOP
(2 + 3)	PUSH 2 PUSH 3 ADD STOP

- Goal: Method parseE should 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 return code for E1 and E2
    - Call these code strings c1 and c2
    - Method returns string c1 + c2 + "ADD\n"
  - Top-level method should tack on a STOP command after code received from parseE

Main Method for Code Gen

```
public static void main (String[] args) {
    String code;              // Holds the generated code
    Scanner lineScanner = new Scanner(System.in);
    String line = lineScanner.nextLine();
    while (line.length() != 0) {
       System.out.println(line);
       Scanner scanner = new Scanner(line);
       try {
          code = parseE(scanner) + "STOP\n";
          if (scanner.hasNext())
             throw new RuntimeException("Line has extra token = " + scanner.next());
       } catch (RuntimeException e) {
          System.err.println(e.getMessage());
          code = "ERROR\n";
       }
       System.out.println(code);
       line = lineScanner.nextLine();
    }
    System.out.println("Quitting");
 }
```

Rest of Code Gen Program

```
public static void check (Scanner scanner, String string) {
      if (!scanner.hasNext()) throw new RuntimeException("Expected more input");
      String token = scanner.next();
      if (token.equals(string)) return;
      throw new RuntimeException("Expected " + string + ", but found " + token);
  }

public static String parseE (Scanner scanner) {
      if (scanner.hasNextInt()) return "PUSH " + scanner.nextInt() + "\n";
      else {
         check(scanner, "(");
         String c1 = parseE(scanner);
         check(scanner, "+");
         String c2 = parseE(scanner);
         check(scanner, ")");
         return c1 + c2 + "ADD\n";
      }
  }
```

Does Recursive Descent Always Work?

- There are some grammars that cannot be used as the basis for recursive descent
  - A trivial example (causes infinite recursion):
    - $S \rightarrow b$
    - $S \rightarrow Sa$
- Can rewrite grammar
  - $S \rightarrow b$
  - $S \rightarrow bA$
  - $A \rightarrow a$
  - $A \rightarrow aA$
- For some constructs, Recursive Descent is hard to use
  - Can use a more powerful parsing technique (there are several, but not in this course)

Syntactic Ambiguity

- Sometimes a sentence has more than one parse tree
  - $S \rightarrow A \mid aaxB$
  - $A \rightarrow x \mid aAb$
  - $B \rightarrow b \mid bB$
  - 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

- This ambiguity actually affects the program's meaning
- How do we resolve this?
  - Provide an extra non-grammar rule (e.g., the *else* goes with the closest *if*)
  - Modify the grammar (e.g., an if-statement must end with a 'fi')
  - Other methods (e.g., Python uses amount of indentation)

Conclusions

- Recursion is a very powerful technique for writing compact programs that do complex things.
- Common mistakes:
  - Incorrect or missing base cases
  - Subproblems must be simpler than top-level problem
- Try to write description of recursive algorithm and reason about base cases etc. before writing code.
  - Why?
    - Syntactic junk such as type declarations... can create mental fog that obscures the underlying recursive algorithm.
  - Try to separate logic of program from coding details.

Exercises

- Think about recursive calls made to parse and generate code for simple expressions
    - 2
    - (2 + 3)
    - ((2 + 45) + (34 + -9))
- Can you derive an expression for the total number of calls made to parseE for parsing an expression?
    - Hint: think inductively
- Can you derive an expression for the maximum number of recursive calls that are active at any time during the parsing of an expression?

Exercises

- Write a grammar and recursive program for palindromes?
    - mom
    - dad
    - i prefer pi
    - race car
    - red rum sir is murder
    - murder for a jar of red rum
    - sex at noon taxes
- Write a grammar and recursive program for strings $A^N B^N$
    - AB
    - AABB
    - AAAAAAABBBBBBB
- Write a grammar and recursive program for Java identifiers
    - <letter> [<letter> or <digit>]$^{0\ldots N}$
    - j27, but not 2j7