Grammars & Parsing

Lecture 5
CS211 – Summer 2007

Application of Recursion

- We have discussed two applications of recursion
  - on integers: Factorial, fibonacci, combinations, a^n
  - on lists: traversing, insertion, deletion, …
- 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.

- 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

- Grammar: set of rules for generating sentences in a language. For example,

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

- A Sentence is 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 valid Sentences:
  - boys see bunnies
  - bunnies like girls
  - …
- The words boys, girls, bunnies, like, see are called *tokens* or *terminals*
- The words Sentence, Noun, Verb are called *nonterminals*
- How big is the set of valid 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

- 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
- How big is the set of valid Sentences?

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?

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.
- The tokens here are the 7 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?
- Here are some legal expressions:
  - 2
  - (3 + 34)
  - ((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 to:
  - define a *language* (i.e., the set of properly structured *sentences*)
  - *parse* a *sentence* (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*

E → integer
E → (E + E)

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*

E → integer
E → (E + E)

Pseudo Code:

```
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

E → integer
E → (E + 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, ")");
}
```


Syntactic Ambiguity

- Sometimes a sentence has more than one parse tree
  S → A | aaxB
  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, e.g. if E1 then if E2 then S1 else S2
- How do we resolve this?
  - Provide an extra non-grammar rule (e.g., the *else* goes with the closest *if*)
  - Modify the language (e.g., an if-statement must end with an 'endif')
  - Operator precedence (e.g. 1 + 2 * 3 should always be parsed as 1 + (2 * 3), not (1 + 2) * 3
  - Other methods (e.g., Python uses amount of indentation)

Detour: Java Exceptions

- Parsing does two things:
  - It returns useful data (a parse tree)
  - It checks for validity (i.e., is the input a valid *sentence*?)
- How should we respond to invalid input?
- *Exceptions* allow us to do this without complicating our code unnecessarily

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 ≥ 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();
}
```


Declaring Exceptions

- In general, any exception that could be thrown 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* the `try` clause of 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

Conclusion

- 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 before writing code
  - Why?
    - Syntactic junk such as type declarations… can create mental fog that obscures the underlying recursive algorithm
  - Best to separate the logic of the 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))
- Derive an expression for the total number of calls made to parseE for parsing an expression
  - Hint: think inductively
- Derive an expression for the maximum number of recursive calls that are active at any time during the parsing of an expression (i.e. max depth of call stack)

Exercises

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