GRAMMARS & PARSING

Lecture 7

CS2110 – Fall 2013

Pointers to the textbook

Parse trees: Text page 592 (23.34), Figure 23-31

- Definition of Java Language, sometimes useful: http://docs.oracle.com/javase/specs/jls/se7/html/index.html
- Grammar for most of Java, for those who are curious: http://csci.csusb.edu/dick/samples/java.syntax.html

Homework:

- Learn to use these Java string methods: s.length, s.charAt(), s.indexOf(), s.substring(), s.toCharArray(), s = new string(char[] array).
- Hint: These methods will be useful on prelim1 (Oct 10)! (They can be useful for parsing too…)

Application of Recursion

- So far, we have discussed recursion on integers
  - Factorial, fibonacci, a^n, combinatorials
- 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://docs.oracle.com/javase/specs/jls/se7/html/index.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
  - …
- 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 *nonterminals*
- 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 last one because the set of Sentences is infinite**

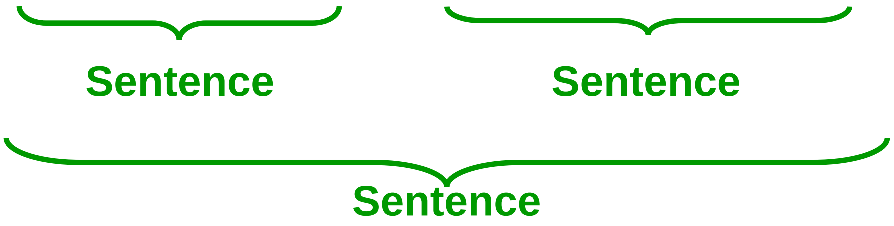
- Examples of Sentences in this language:
  - boys like girls
  - boys like girls and girls like bunnies
  - boys like girls and girls like bunnies and girls like bunnies
  - boys like girls and girls like bunnies and girls like bunnies and girls like bunnies
  - ………

- 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 → Sentence or Sentence .
- 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

- PunctuatedSentence → 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 (.)
- This grammar is ambiguous:

boys like girls
and girls like boys
or girls like bunnies

Uses of Grammars

- Grammar describes every possible legal expression
  - You could use the grammar for Java to list every possible Java program. (It would take forever)

- Grammar tells the Java compiler how to understand a Java program

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)))

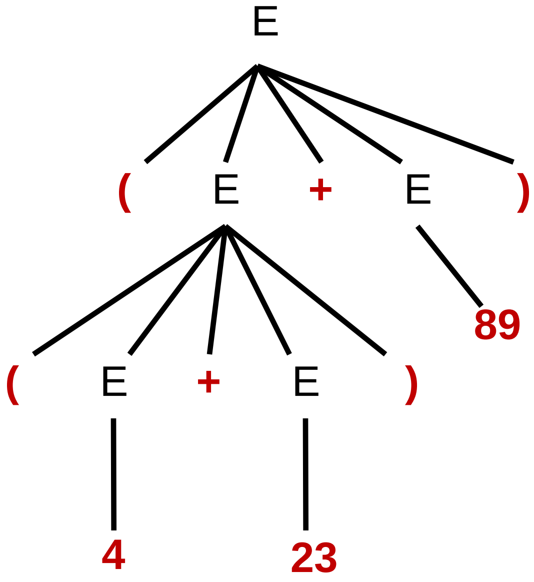
- Here are some illegal expressions:
  - (3
  - 3 + 4

- 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*)**
- 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!

```
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 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);
}
```

Detour: Error Handling with 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 ε the length of the array

- In our case (parsing), we should throw an exception when the input cannot be parsed

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

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 for E1 and E2 return code strings c1 and c2, respectively
    - Then to compile (E1 + E2), return c1 + c2 + "ADD\n"
  - Top-level method should tack on a STOP command after code received from parseE

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 → b**
    - **S → Sa**
- Can rewrite grammar
    - **S → b**
    - **S → bA**
    - **A → a**
    - **A → 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 → 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**
- **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?
- 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 a 'fi')
- Operator precedence (e.g.
  1 + 2 * 3 should be parsed as
  1 + (2 * 3), not
  (1 + 2) * 3
- Other methods (e.g., Python uses amount of indentation)

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, etc. 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 sentence palindromes that ignores white spaces & punctuation
  - **Was it Eliot's toilet I saw?**
  - **No trace; not one carton**
  - **Go deliver a dare, vile dog!**
  - **Madam, in Eden I'm Adam**
- 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**