Grammars and Parsing

Lecture 5
CS211 – Fall 2005

Announcements

- CMS is being "cleaned"
  - If you did not turn in A1 and haven't already contacted David Schwartz, you need to contact him *now*
- Java Reminder
  - Any "significant" class should be declared *public* and should appear in a file whose name matches the class name
- Academic Integrity
  - There were some AI violations on Assignment 1
  - We treat such violations seriously and have begun the hearing process
  - We test all pairs of submitted programming assignments for similarity
    - Similarities are caught even if variables are renamed

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: Grammars and 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	→	dogs
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 'dogs'
  - A Verb can be 'like' or 'see'
- Grammar: set of rules for generating sentences in a language
- Examples of Sentence:
  - boys see dogs
  - dogs 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	→	dogs
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 dogs
  - boys like girls and girls like dogs and girls like dogs
  - boys like girls and girls like dogs and girls like dogs and girls like dogs
  - ………
- 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 dogs . .

Sentences with Periods

TopLevelSentence → Sentence **.**
Sentence → Sentence and Sentence
Sentence → Sentence or Sentence
Sentence → Noun Verb Noun
Noun → boys
Noun → girls
Noun → dogs
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

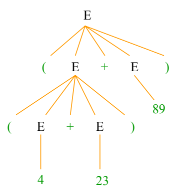
E → integer
E → (E + E)

- This is a grammar for 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, ")");
}
```

A Note on Boolean Operators

- Java supports two kinds of Boolean operators:
  - E1 & E2:
    - Evaluate both E1 and E2 and compute their conjunction (i.e.,“and”)
  - E1 && E2:
    - Evaluate E1. If E1 is false, E2 is not evaluated, and value of expression is false. If E1 is true, E2 is evaluated, and value of expression is the conjunction of the values of E1 and E2.
- In our parser code, we use &&
  - If *check(scanner, “)”)* returns false, we simply return false without trying to read anything more from input
    - This gives a graceful way to handle errors

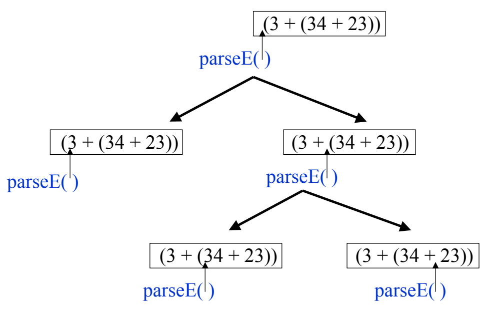
Helper Method

```
public static boolean check (Scanner scanner, String string) {
    if (!scanner.hasNext()) {
       System.err.println("Missing token: " + string);
       return false;
    }
    String token = scanner.next();
    if (!token.equals(string)) {
       System.err.println("Expected " + string + ", but found " + token);
       return false;
    }
    return true;
  }
```

Main Program

```
public static void main (String[] args) {
     Scanner sc = new Scanner(System.in);
     String line = sc.nextLine();
     while (line.length() != 0) {
        System.out.println(line);
        Scanner scanner = new Scanner(line);
        boolean valid = parseE(scanner);
        if (valid && scanner.hasNext()) {
           valid = false;
           System.err.println("Extra token: " + scanner.next());
        }
        System.out.println(valid? "Valid" : "Invalid");
        line = sc.nextLine();
     }
     System.out.println("Exiting");
  }
```

Trace of Recursive Calls to parseE



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
```

Idea

- Recursive method parseE should return a string containing stack code for expression it has parsed
- Top-level method should tack on a STOP command after code received from parseE
- Method parseE generates 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
      - Say these are strings c1 and c2
    - Method returns c1 + c2 + “ADD\n”

Main Program

```
public static void main (String[] args) {
    String code = null;
    Scanner sc = new Scanner(System.in);
    String line = sc.nextLine();
    while (line.length() != 0) {
        System.out.println(line);
        Scanner scanner = new Scanner(line);
        try {
            code = parseE(scanner) + “STOP\n”;
            if (scanner.hasNext()) error("Extra token: " + scanner.next());
        } catch (RuntimeException e) {
            code = “ERROR\n”;
        }
        System.out.println(code);
        line = sc.nextLine();
    }
    System.out.println("Exiting");
}
```

Rest of Code Gen Program

```
public static void error (String message) {
    System.err.println(message);
    throw new RuntimeException();
}
public static void check (Scanner scanner, String string) {
    if (!scanner.hasNext()) error("Missing token: " + string);
    String token = scanner.next();
    if (!token.equals(string)) error("Expected " + string + ", but found " + token);
}
public static String parseE (Scanner scanner) {
    if (scanner.hasNextInt()) return "PUSH " + scanner.nextInt() + "\n";
    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 -> b
    - S -> Sa
- Can rewrite grammar
  - S -> b
  - S -> bA
  - 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
  - $S \to A \mid aaB$
  - $A \to \varepsilon \mid aAb$
  - $B \to \varepsilon \mid aB \mid bB$
  - The string aabb 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)

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

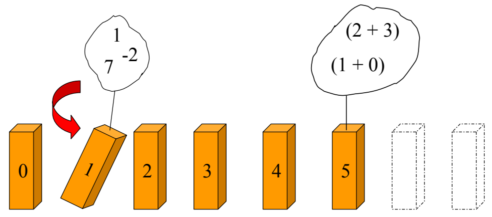
- Write a grammar and recursive program for palindromes?
  - mom
  - dad
  - i prefer pi
  - race car
  - red rum sir is murder
  - murder for a jar of red rum
  - sex at noon taxes
- Write a grammar and recursive program for strings $A^N B^N$
  - AB
  - AABB
  - AAAAAAABBBBBBB
- Write a grammar and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

Number of Recursive Calls

- Claim:
  - # of calls to parseE for expression E =
    - # of integers in E + # of addition symbols in E.
- Example: ((2 + 3) + 5)
  - # of calls to getExp = 3 + 2 = 5

Inductive Proof

- Order expressions by their length (# of tokens)
- E1 < E2 if length(E1) < length(E2).



Proof of # of recursive calls

- Base case: (length = 1)
  - Expression must be an integer
  - parseE will be called exactly once as predicted by formula
- Inductive case: Assume formula is true for all expressions with *n* or fewer tokens.
  - If there are no expressions with *n*+1 tokens, result is trivially true for *n*+1
  - Otherwise, consider expression E of length *n*+1. E cannot be an integer; therefore it must be of the form (E1 + E2) where E1 and E2 have *n* or fewer tokens. By inductive assumption, result is true for E1 and E2.

```
#-of-calls-for-E     = 1 + #-of-calls-for-E1 + #-of-calls-for-E2
                     = 1 +      #-of-integers-in-E1 + #-of-'+'-in-E1 +
                                #-of-integers-in-E2 + #-of-'+'-in-E2
                     = #-of-integers-in-E + #-of-'+'-in-E
   as required
```

Conclusion

- Recursion is a very powerful technique for writing compact programs that do complex things.
- Common mistakes:
  - Incorrect or missing base cases
  - Sub-problems 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.