Recursion and Recursive Descent Parsing

CS211
Fall 2000

Divide & Conquer Outline

- D & C Outline

```
public Solution DaC (Problem P) {
    if (P is small)
        return solution for P;
    Break P into parts P1 and P2;
    DaC(P1); DaC(P2);
    Use the solutions for P1 and P2
        to produce a solution for P;
    return solution for P;
    }
```

- QuickSort

```
private static void quickSort
        (int[ ] A, int low, int high) {
    if (low < high) {
        int p = partition(A,low,high);
        quickSort(A,low,p – 1);
        quickSort(A,p,high);
        }
    }
```

2

Recursion vs. Induction

Lemma 1 The partition method splits A[low..high] into two groups: those ≤ the pivot and those ≥ the pivot.

Proof Based on the invariant: A[low..i–1] ≤ pivot & A[j+1..high] ≥ pivot

Theorem QuickSort correctly sorts any array of int.

Proof By Lemma 2 it works correctly on the subarray from 0 to length-1 (which is the entire array).

Lemma 2 QuickSort correctly sorts any subarray of int.

Proof

Basis: It works correctly on a subarray of size 1.

Induction Hypothesis: It works correctly on a subarray of size k<n.

For a subarray of size n, partition works (by Lemma 1) and splits the subarray into two smaller pieces. By the induction hypothesis these pieces are sorted correctly. These smaller pieces are in the correct order in relation to each other, so the subarray of size n is correctly sorted.

3

A Parsing Example

- The goal is to parse (and evaluate) a simple boolean expression (BE)
- (Recursive) Definition:
  - The constants T and F are BEs
  - If E is a BE then !E is a BE
  - If E and F are BEs then so are (E & F), (E | F), and (E = F)
- BE Examples
  - !(T=F)
  - (!F & !T)
  - ((F & !F) & F)
  - (F | !(T & F))
- HW3 is a similar task

4

Lexical Analysis

- We assume that we have a lexical analyzer
  - A lexical analyzer (or *tokenizer*) divides the input stream into *tokens*
- The tokenizer has the following methods
  - nextToken( ): return the next token from input
  - pushBack( ): push a token back so it can be retrieved again by nextToken( )
- A *token* is a single, simple unit of a language
- In Java, tokens are
  - *keywords* (e.g., this, null, if, while),
  - *identifiers* (e.g., i, count),
  - *numbers* (e.g., 0, 1.5, 6.02e23),
  - *strings*,
  - *operators* (e.g., +, <=, !=),
  - …
- For our example, a token is a single (nonblank) char

5

A Recursive Descent BE Evaluator

```
class BooleanExp {
    Tokenizer in;

    public BooleanExp (String input) {
        in = new Tokenizer(input);
        }

    public boolean evaluate ( ) {
        boolean answer = be( );
        if (in.hasMoreTokens()) error;
        return answer;
        }

public boolean be ( ) {
    char ch = in.nextToken( );
    if (ch == 'T') return true;
    if (ch == 'F') return false;
    if (ch == '!') return !be( );
    if (ch == '(') {
        boolean left = be( );
        char op = in.nextToken( );
        boolean right = be( );
        if (in.nextToken( ) != ')') error;
        if (op == '&') return left & right;
        if (op == '|') return left | right;
        if (op == '=') return left == right;
        error;
        }
    error;
    }}
```

6

Errors While Parsing

- Desired responses to a parsing error
  - Produce error message
  - Recover and continue parsing
- Recovery depends on finding an "understandable" token (e.g., ";" or "eol")
- Exceptions make it easier to handle parsing errors

```
if (ch == '(') {
    boolean left = be( );
    char op = in.nextToken( );
    boolean right = be( );
    if (in.nextToken( ) != ')') throw new
        IllegalArgumentException("Missing ')'");
    if (op == '&') return left & right;
    if (op == '|') return left | right;
    if (op == '=') return left == right;
    throw new
        IllegalArgumentException("Bad op");
    }
```

7

Catching the Parsing Exceptions

- The try/catch construction allows the errors to be handled without cluttering the code
- Without try/catch:
  - Code has many if/else branches
  - What do you return to indicate an error?

```
try {
    BooleanExp b = new BooleanExp(string);
    System.out.println (b + " is " + b.evaluate());
    }
catch (NoSuchElementException e) {
    System.out.println("Incomplete expr");
    }
catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
    }

// For this example, NoSuchElementException
// is thrown by the Tokenizer when it
// unexpectedly runs out of tokens;
// IllegalArgumentException is thrown when an
// unexpected token occurs.
```

8

More Complicated Expressions

- We haven't used pushBack( ); is it really needed?
- Suppose we want more realistic Boolean Expressions
  - T & (T|!F) & !F & (T|F)

We distinguish between BTerms and BExps

- The constants T and F are BTerms
- If S is a BTerm then so is !S
- If E is a BExp then (E) is a BTerm
- A BExp is one or more BTerms separated by &, |, or =
- The operators &, |, and = are left-associative

9

Left- vs. Right- Associativity

- Many operators are associative
  - (5+3)+2 is the same as 5+(3+2)
  - (5*3)*2 is the same as 5*(3*2)
- Other operators are *not* associative
  - (5-3)-2 is different from 5-(3-2)
  - (5/3)/2 is different from 5/(3/2)
- A rule is needed for when parentheses are not present
  - Left-associative implies group starting from the left {e.g., 5-3-2 is treated as (5-3)-2}
  - Right-associative implies group starting from the right {e.g., 2^3^2 is treated as 2^(3^2)}
- This is separate from any precedence rules

10

Using pushBack( )

```
public boolean bexp ( ) {
    boolean result = bterm( );
    char ch = in.nextToken( );
    while (ch == '&' | ch == '|' | ch == '=') {
        if (ch == '&') result = result & bterm( );
        if (ch == '|') result = result | bterm( );
        if (ch == '=') result = result == bterm( );
        ch = in.nextToken( );
        }
    in.pushBack( );     // Not an op so it's not ours
    return result;
    }
```

- Parsing is easiest if each routine is carefully designed to process only its own tokens
- Note that operations are done from left-to-right

11