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);
      }
    }

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)

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 Recursive Descent BE Evaluator

- 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

- A boolean expression (BE) consists of:
  - A boolean constant (T, F)
  - An operator (&, |, =)
  - Parentheses

- For example, a boolean expression could be:
  - (T & F)
  - !F
  - (T | F)
  - !(T & F)
  - (T = F)

A Recursive Descent BE Evaluator

- A boolean expression (BE) consists of:
  - A boolean constant (T, F)
  - An operator (&, |, =)
  - Parentheses

- For example, a boolean expression could be:
  - (T & F)
  - !F
  - (T | F)
  - !(T & F)
  - (T = F)
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

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?

More Complicated Expressions

- We haven’t used pushBack(); is it really needed?
- Suppose we want more realistic Boolean Expressions
  - $T \& (T|F) \& IF \& (T|F)$
- We distinguish between BTerms and BExps
  - The constants T and F are BTerms
  - If $S$ is a BTerm then so is $\neg S$
  - If $E$ is a BExp then $\{E\}$ is a BTerm
  - A BExp is one or more BTerms separated by $\&$, $\|$,
    and $\pm$
  - The operators $\&$, $\|$, and $\pm$ are left-associative

Left- vs. Right- Associativity

- Many operators are associative
  - $(5+3)+2$ is the same as $5+(3+2)$
  - $(5\times3)^2$ 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

Using pushBack()

```java
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