Finish with Recursion

CS211
Fall 2000

Another Recursive Descent Example

- Goal: Determine if the brackets ( ) [ ] { } on a line are balanced and properly nested
- Examples:
  - legal: ( ) [( )]
  - legal: ( ) { } [ ]
  - illegal: ([ )]
  - illegal: (})
- Recursive definition for LegalExp
  - The empty string is a LegalExp
  - If E is a LegalExp then so are (E), [E], and {E}
  - One or more LegalExps on a line make a LegalExp

Step 1: Build a Tokenizer

- We want to divide an input line (a String) into tokens
  - The Java API on the Web includes a java.util package
  - This package contains java.util.StringTokenizer
- StringTokenizer has a nextToken( ) method that throws NoSuchElementException if it runs out of tokens
- The StringTokenizer constructor takes
  - an input string,
  - a list of delimiters (token separators), and
  - a boolean flag (true implies that delimiters are also tokens)
- StringTokenizer also has other methods
  - countTokens( )
  - hasMoreTokens( )
  - a couple others

Using StringTokenizer

- Can StringTokenizer be used directly? Almost…
  - We want to skip all "uninteresting" tokens
  - Would like an eol token
- We can alter StringTokenizer to do these extra things by using either
  - inheritance or
  - aggregation
- Inheritance:
  - We make our own tokenizer by *extending* StringTokenizer
  - Existing methods are either inherited or overridden
- Using inheritance:
  - Either we override all methods of StringTokenizer
  - Or we accept some inherited methods with "surprising" behavior

Tokenizer Code

- Aggregation
  - We make our own tokenizer by using a StringTokenizer *within* our tokenizer class
  - Our class has only the methods that we choose to write
    - nextToken()
    - pushBack()

```
class MyTokenizer {
    private StringTokenizer tokenizer;
    private String currentToken;
    private boolean pushed;
    public static String PARENS = "()[]{}";

    public MyTokenizer (String inputString) {
        tokenizer = new
StringTokenizer(inputString,PARENS,true);
        currentToken = null;
        pushed = false;
        }

    public void pushBack () {
        pushed = true;
        }
```


nextToken()

```
    public String nextToken ( ) {
        if (pushed) {
            pushed = false;
            return currentToken;
            }
        try {
            do {
                currentToken = tokenizer.nextToken();
                } while (PARENS.indexOf(currentToken) == -1);
            } catch (NoSuchElementException e) {
                currentToken = "eol";
            }
        return currentToken;
        }
}
```

Step 2: Build a Parser

- Follow the recursive definition
  - The empty string is a LegalExp
  - If E is a LegalExp then so are (E), [E], and {E}
  - One or more LegalExps on a line make a LegalExp

```
class LegalExpParser {
    MyTokenizer in;
    public void legalExp () {
        String matcher;
        String token = in.nextToken();
        while ("({[".indexOf(token) != -1) {
            if (token.equals("(")) matcher = ")";
            else if (token.equals("[")) matcher = "]";
            else matcher = "}";
            legalExp();
            token = in.nextToken();
            if (!token.equals(matcher)) error;
            token = in.nextToken();
            }
        in.pushBack();
        }
```

7

The Rest of the Code

```
public void eval (String inputString) {
  in = new MyTokenizer(inputString);
  legalExp();
  if (in.nextToken().equals("eol")) return;
  error;
  }
} // end LegalExpParser

  // Code using a LegalExpParser
  // The String string is to be evaluated for brackets
  LegalExpParser p = new LegalExpParser();
  try {
    p.eval(string);
    System.out.println(string + " is OK");
    } catch (IllegalArgumentException e) {
      System.out.println(e.getMessage());
    }
```

8

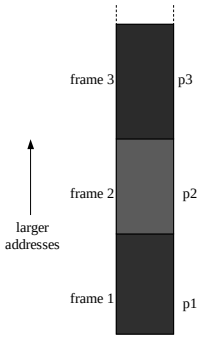
How Recursion Works

```
int fact  (int n) {
    if (n==0) return 1;
    else return n*fact(n-1);}
```

- How do we prevent the different 'n's from overwriting each other?
  - Give them different locations
- How do we know which 'n' to compute with at each moment?
  - Organize the 'n's into a stack; always use the topmost value
- How does a method-call know where to return to?
  - Save a *return address* before making the call
- How does the *callee* return the result to the *caller*?
  - Result is left on top of the stack
- How do we keep track of all this information?
  - Organize the information for each call into a *frame*

9

Stack Frames



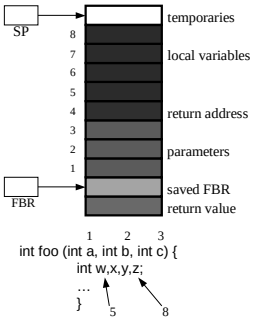
p1 calls p2 which calls p3

- FBR (Frame Base Register) points into current (topmost) frame
- Values within frames are addressed via offsets from FBR
- SP (Stack Pointer) moves up and down as temporary values are pushed and popped

10

Typical Stack Frame

- One slot for each parameter (values computed and pushed by *caller*)
- One slot for return address (where to continue execution after method is done)
- One slot for each local variable (allocated by *callee*)
- Temporaries are pushed/popped on top of frame



11

How Exceptions Work

- What a Java program does when an exception occurs

  while *not* within a *try* that has a matching *catch* for this exception {
  Pop the top frame off the stack;
  }
  Execute the *first* matching *catch*-block;
  Execute the *finally* block if there is one;
  Continue with following code;

- When running a Java application
  - There is a "first" frame that catches all exceptions
  - It prints some semi-helpful information and then halts

12