Recursive Descent Parser for Java GUIs

Zachary Smith

25 September 2019

CMSC 330 – Advanced Programming Languages

Minute changes in a context-free grammar’s formations are difficult to square with an object-oriented language and mindset. Each production listed in Project1.pdf is represented by an object of a subclass of ParseTreeNode like Gui, Layout, LayoutType, etc. By the grammar rules, a sequence of gui components will be represented by a severely imbalanced tree structure as each new Widget or RadioButton requires allocating two more objects. Tokens are stored in an ArrayList<String>, and an iterator is created to pass down the call stack so Gui.parse(itr) calls Layout.parse(itr) and Widgets(itr), ad nauseum. Super class ParseTreeNode has string-matching methods that use regex and take any opportunity to advance the iterator – some are better than others. After finishing Gui.parse I wrote new string-matching methods with a clearer focus on operators like parentheses and commas, but that had less strict matches and became too greedy with tokens that had to pass all the way down the call stack consistently. Widget.parsePanel and Widget.parseGroup needed their own internal loops as I could not find a way to reliably match that far down the call stack.

I have a hunch that a language like Java that enforces object-oriented thinking is not as suited to this task as a functional language. Each subclass of ParseTreeNode has a String array of valid keywords, and allocating a child object makes it convenient to check a token against those keywords before parsing, but this led me to recursive behavior that seems embarrassing in retrospect. Essentially:

class Widgets {

Widget childW;

public Widgets(ListIterator<String> itr) {

this.itr = itr;

this.childW = new Widget(itr);

} //…

}

class Widget {

Widgets childWs;

public Widget(ListIterator<String> itr) {

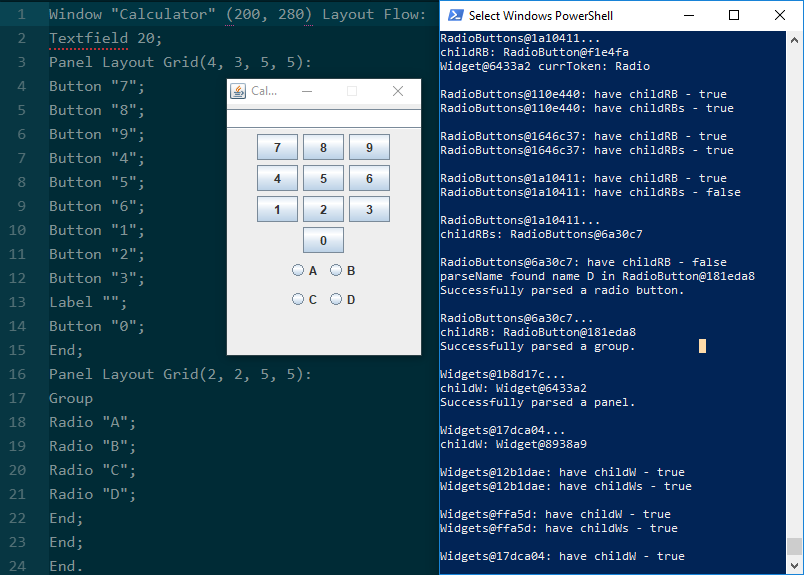
this.itr = itr;

this.childWs = new Widgets(itr);

} //…

}

At the time there was more code obscuring the issue, but once a single instance of either class is allocated, it leads to unbounded recursion as each constructor continually calls on the other. To avoid going back to edit old code again, I defined each validKeywords array as having the keywords for that tree node and those of its descendants. Declaring the array static could accomplish the same goal, but I had earlier issues with that approach and tabled it.

Each method has been tested thoroughly for individual accuracy, but after days of debugging complications arose from the *relationships* between methods, how they affect the iterator or match a keyword or pass a GUI component. String matching functions take whitespace in to account, and are tolerant of adjacent tokens *to a degree*. The method parseName is only forgiving of trailing semicolons, not parentheses. Also, since the goal of the parse methods is to build a tree, throwing ParseError may lead to unexpected behavior as the tree still exists and will try to build a GUI with what information it has. That being said, all components work as advertised and they properly recurse (no endless loops).

I took several hand-written notes during development. Here is the pseudocode I wrote after finishing building a parse tree and planning to add GUI functionality:

