Summary and evaluation

The pearl calculates a function that takes a list of brackets and finds the a longest contiguous subsequence in which brackets are balanced. The function that is calculated runs in linear time. (I can't decide if I should be surprised that the longest balanced sequence can be found in linear time.) The calculation draws key ideas from LR parsing (reversed), and it builds on the converse-of-a-function theorem to express a parser as a fold. Converting the fold to a scan gives the linear-time behavior.

I judge a functional pearl by these criteria (Bird, 2006):

> A functional pearl is meant to be polished, elegant, instructive, and entertaining. Ideal pearls are "grown from real problems that have irritated programmers." Pearls contain:
>
> - Instructive examples of program calculation or proof;
> - Nifty presentations of old or new data structures;
> - Interesting applications and programming techniques.

This submission was not able to hold my interest. Here are some elements I found disappointing:

- The problem has deep connections with parsing, but those connections are not elucidated to my satisfaction.
- The treatment of partially parsed trees is more complicated than I had hoped for.
- During the detours of sections 3 and 4, I lost track of the big picture.
- The submission takes a roundabout path from inputs, to trees, to spines, back to inputs. But I think a lot could be done with inputs directly. See my suggestion below, which I think renders a lot of the development here superfluous.
- The introduction suggests that the choice of grammar/representation is arbitrary. But I'm not at all sure that the development works with any other choice—and on this topic, the submission is silent.
- The subtitle of the submission suggests a focus on program inversion. The actual delivery gives a limited example of a limited form of program inversion. I don't think it delivers on the promise implied by the title.
- The proof in Appendix A is not a good advertisement for equational reasoning.

Suggestion for the author

To cut through a lot of the development here, consider what is a prefix [suffix] of a properly bracketed sequence. It is a sequence of "items" wherein each item is either a left [right] bracket or a properly bracketed sequence. The suffix is your spine; here are some possible representations:

```
data Item = Tree Tree | Right
type Spine = [Item]

----  { since a Tree can be empty, we can make them alternate }

data Right = Right
type Spine = ([(Tree, Right)], Tree)

----  { make the right bracket implicit }

type Spine = ([Tree], Tree)

----  { make "nonempty" a dynamic invariant, not enforced by the type }

type Spine = [Tree]
```

It might now be useful to define these functions:

```
consRight :: Spine -> Spine
consLeft  :: Spine -> Spine

consRight trees = Null : trees
consLeft  (t1 : t2 : ts) = makeTree Left t1 Right t2 : ts
consLeft  _ : _ = [Null] -- unbalanced; must start over
```

This development eliminates the choice on lines 20 and 23 of your submission, which is presented as arbitrary. That representation is now revealed as *necessary* because it is the natural representation that supports the `consLeft` operation.

I believe that if each tree is annotated with its length and each spine is annotated with the length of its longest tree, then the entire problem can be solved by `foldr`:

```
lbp = best . foldr cons null
   where cons '(' = consLeft
         cons ')' = consRight
         null = annotate [Null]
```

Detailed comments for the author

Since the task you pose is it not inherently interesting, I'd like your paper to open with some statement about what we can expect to find pearly in the presentation below. Is the main accomplishment to solve the task in linear time?

On line 20, I'm curious about what happens if you choose a different grammar. What happens to your development?

On line 29, a type for function `lbp` would not be out of place.

I can't help but think that functions `pr` and `parse` go together. Having them separated by the formal statement of the problem is not helping. Especially since you can't read the problem statement until you know about `parse`.

The statement on line 35 is a little misleading. While it is technically correct that `inits` produces the prefixes of *its* input list, the meaning of the phrase "input list" changes from place to place on line 31. I could easily think you meant the list that is input to the *composition.* I would rather read that `tails` computes all suffixes of the input list and then `map inits` computes all the prefixes of those suffixes.

On line 66, I don't know why the next step is usually to apply the scan lemma. What's usual about the scan lemma? Would that happen in all derivations? Is it because the goal is a linear-time solution? Is it because of particular problem you've chosen?

Lines 80 and 81 are the first place where I get a hint about what might make your paper interesting. That's too long to make me wait.

I was confused by the typographic choices starting on line 129. I'd like that presentation to open with the idea that you've got metavariables t, u, and v that stand for trees. Please show us those metavariables in italics. Then when you present the string used to construct the tree in Figure 1, write those metavariables in italics even in the string itself.

"Balanced brackets" is a problem that screams "parsing" and "pushdown automata." So at the beginning of section 3, the obvious game is LR parsing with a stack. I suppose you could put an analogous right-to-left parsing also with a stack. But some relationship with standard stuff on parsing would be appreciated here.

I recognize that you have a problem: the standard literature on parsing works from left to right, but the standard literature on folds works from right to left. This is nobody's fault, but in order to help readers overcome this inconsistency, I expect you to work extra hard.

On line 137, "along" is misspelled as "alone."

I think the presentation of the spine should be followed immediately by the presentation on lines 148 to 154. That way we can be confident that we understand what the spine data structure represents. Before you show how a spine is converted to a tree, show us how the spine is rendered as a string.

I'd also really like to see a parser that just uses an explicit stack. I think that would make everything clearer.

Starting on line 193, I'd like to see this analogy developed in much greater depth. Since what you have is effectively a parsing problem, the parallels deserve to be made explicit. I'm not sure where is the right place to develop this analogy, but earlier might have been better.

On line 224 the name of the function `unwrapM` is misspelled.

As we get to section 4, I'm losing interest.

On line 244, it seems to me that we are now leaving principled derivation behind.

On line 257, if I understand what's going on, the use of `build` *does* generate something new: it generates a list of null spines. These look harmless, as a null sequence is always balanced, and `maxBy` will discard duplicates.

Appendix A looked gnarly. It's not a good fit for a pearl.

On lines 264 to 275, I am not following the development easily, and I am not motivated to work hard to follow it. Is your ordering going to replace the composition of `size` and `unwrap`, or what?

On lines 302, 304, 308, and 312, I'm not sure I trust the application of `head` to the result of `maxBy`. I'm having trouble reconciling that with lines 301, 29, and others.

On lines 325 to 328, this reveal comes quite late. It probably belongs in the very first paragraph, not the very last.