15–16 (*language*) The title and the remainder use the terminology of segments being ⌈balanced⌉, but only the abstract and this sentence use ⌈properly bracketed⌉. For both occurrences: just replace ⌈ properly bracketed ⌉ by: ⌈ balanced ⌉.

21 (*clarity*) Most readers will be familiar with the concept of LL(1) grammars. Consider replacing ⌈unambiguous⌉ by ⌈LL(1) and therefore ambiguous⌉.

27 (*clarity*) An example will help the reader check their understanding of these definitions. Add: ⌈ For example, the string `"(()())()"` is represented by the tree `Fork (Fork Null (Fork Null Null)) (Fork Null Null)`, and indeed, this is the result of applying function *pr* to this tree. ⌉.

28 (*clarity*) The components *maxBy size* and *segments* in the following specification of function *lbp* hold no surprise: the first speaks for itself, and the second should be well known. The role of $filtJust$, in contrast, can only be understood in combination with the meaning of *parse*, specifically how its partiality is represented through "monadification". It would therefore seem better to move the explanation of function *parse* up, from lines 35–40 to before line 28. Because *parse* is best explained as an implementation of pr^{-1}, it is better to discuss this inverse first. This requires some reshuffling and further adjustments of the text, for which I suggest:

> Function *pr* is injective but not surjective: it does not yield unbalanced strings. Therefore its right inverse, that is, the function pr^{-1} such that $pr(pr^{-1}\ xs) = xs$, is partial; its domain is the set of balanced parenthesis strings. We implement it by a function that is made total by using the Maybe monad. This function *parse* :: `String` → `Maybe Tree` builds a parse tree – *parse xs* should return `Just` t such that $pr\ t = xs$ if xs is balanced, and return `Nothing` otherwise. We will construct parse more formally in Section 3.

Then replace ⌈ thus ⌉ by: ⌈ then ⌉.

47 (*language*) Replace ⌈ chooses only those elements ⌉ by: ⌈ collects the elements (of type a) ⌉.

54 (*clarity*) Add: ⌈ Here and in the following, "optimal" means: balanced, and of maximal length. ⌉.

74 (*language*) Replace ⌈ turn $maxBysize \cdot filtJust \cdot map\ parse \cdot inits$ into a $foldr$ ⌉ by: ⌈ express $maxBysize \cdot filtJust \cdot map\ parse \cdot inits$ in the form $foldr\ (\oplus)\ e$ ⌉.

75 (*language*) Replace ⌈ Since *inits* is a $foldr$ — ⌉ by: ⌈ Since *inits* can be expressed as a right fold, given by ⌉.

77–78 (*language*) Replace ⌈fuse ... into ...⌉ by: ⌈fuse ... with ...⌉.

79 (*language*) Replace ⌈ *parse* shall be a $foldr$ ⌉ by: ⌈ *parse* needs to be a right fold ⌉.

80 (*language*) Replace ⌈ Since *parse* is defined in terms of pr^{-1}, it would be helpful ⌉ by: ⌈ Since *parse* implements pr^{-1}, it will be helpful ⌉.

91 (*typo; language*) Replace ⌈ $filtJust$ is called $catMaybe$ in the standard library ⌉ by: ⌈ $filtJust$ is called $catMaybes$ in the basic Haskell libraries ⌉. (Note the plural form $Maybe\underline{s}$.)

104 (*language*) Replace ⌈ for our needs ⌉ by: ⌈ to our needs ⌉.

105 (*language*) Replace ⌈ when t is a list ⌉ by: ⌈ for the case where t is a list type ⌉.

107–112 (*clarity*) Why use the names *base* and *step* instead of e and $(\oplus)$? The latter will make this and the following developent more readable. (For *bstep* in function *build*, then use e.g. $(\otimes)$.)

111 (*error*) Replace ⌈ $foldr\ step\ base\ xs$ ⌉ by: ⌈ $foldr\ step\ base$ ⌉.

112 (*error*) Replace ⌈ *for all* xs *in the domain of* f^{-1} ⌉ by: ⌈ *for all* xs *in the range of* f ⌉.

121 (*typo*) Replace ⌈ input are ⌉ by: ⌈ inputs are ⌉.

122 (*language*) Replace ⌈ that print ⌉ by: ⌈ trees that print ⌉.

123 (*clarity*) Replace ⌈ t should represent some partially built trees that can still be extended from the left ⌉ by: ⌈ t should also be able to represent partially built trees that can still become balanced by being extended to the left ⌉.

124 (*clarity*) The clause ⌈, while *step* x t ... additional xs⌉ can be left out; it has no function but merely repeats a clear and concise equation in a more clumsy and verbose way.

130 (*language*) Replace ⌈ extended from left ⌉ by: ⌈ extended to the left ⌉.

130 (*clarity*) Replace ⌈ `"(()(()t)u)v"` ⌉ by: ⌈ `"(()(()t)u)v"`, where `t`, `u` and `v` stand for segments of balanced parentheses ⌉.

130 (*clarity*) Replace ⌈ the three trees t, u, and v under the dotted line ⌉ by: ⌈ the three trees t, u, and v under the dotted line,¿¿ where $pr(t)$ = `"t"`, $pr(u)$ = `"u"`, and $pr(v)$ = `"v"`⌉.

131–135, 190–194, 206, 227 (*language*) In an exposition of this kind, where the audience is taken on a guided tour, the pronoun ⌈we⌉ generally stands for ⌈the author(s), together with their audience⌉. Here, however, it suddenly stands for something entirely different: the process executing a program under development, as if the authors and their audience have turned into computing machinery. While this abuse of language is usual in a lecture, it should be avoided in writing. It is also potentially ambiguous, for example in the question ⌈How do we print a spine?⌉ (line 149), where it asks how to code the printing process, but could be read as asking how printing code will be executed.
Replace ⌈ we have / we should / we read / we start / ... ⌉ by: ⌈ the process has / the process should / the process reads / the process starts / ... ⌉. Of course, creative variations are possible and desirable, such as using the passive voice (⌈when only `")t)u)v"` has been read⌉) or being more specific ⌈the parsing process starts⌉.

136 (*clarity*) Replace ⌈ list of trees ⌉ by: ⌈ list of (fully built) trees ⌉. (We now also have partially built trees.)

160 (*clarity*) Replace ⌈ an inductive definition of prS that does ⌉ by: ⌈ a new definition for prS, one that is inductive and does ⌉.

165 (*clarity*) Replace ⌈ definitions of pr and prS ⌉ by: ⌈ definition of pr, original definition of prS ⌉.

167 (*clarity*) Replace ⌈ definition of prS ⌉ by: ⌈ original definition of prS ⌉.

170 (*clarity*) Replace ⌈ following definition of prS ⌉ by: ⌈ following new definition of prS ⌉.

189, 191 (*punctuation*) Replace ⌈ , thus ⌉ by: ⌈ ; thus, ⌉.

193 (*language*) Replace ⌈ Some readers might have noticed ⌉ by: ⌈ Readers may have noticed ⌉. (Or, just ⌈Notice⌉.)

207 (*language*) Replace ⌈ notice ⌉ by: ⌈ recall ⌉.

209 (*punctuation*) Replace ⌈ , otherwise ⌉ by: ⌈ ; otherwise, ⌉.

224 (*typo*) Replace ⌈ *unwarp* ⌉ by: ⌈ *unwrap* ⌉.

227 (*clarity*) In this case, the parentheses around the consed expression actually make this less readable. Replace ⌈ (`'(`s : xs) ⌉ by: ⌈ `'(`: xs ⌉.

227 (*clarity*) Replace ⌈ when the recursive call returns $[t]$ ⌉ by: ⌈ if the recursive call has returned a singleton list $[t]$ ⌉.

229 (*punctuation*) Replace ⌈ while if ⌉ by: ⌈ while, if ⌉.

251 (*error*) Replace ⌈ *build* and $parseS$ ⌉ by: ⌈ *build* ⌉. Note that the functions have different result types and cannot return the same result.

251–252 (*clarity*) Replace ⌈ $parseS$ is a partial function ⌉ by: ⌈ $parseS$ implements a partial function ⌉.

252–253 (*clarity*) Replace ⌈ *build* is a total function that parses a prefix of the string. ⌉ by: ⌈ *build* is a total function that parses the maximal prefix of the string that can still become balanced by being extended to the left. ⌉.

254–282 (*proof structure*) Let *opref* xs denote the optimal prefix of xs. The following three related propositions hold.

(1) If $parseS\ xs = \mathsf{Just}\ s$, then $build\ xs = s$.
(2) $parseS\ xs = \mathsf{Just}\ [t]$ iff $pr\ t = xs$.
(3) $head\ (build\ xs) = t$ iff $pr\ t = opref\ xs$.

A weaker version is found in the first paragraph :w of the appendix. It seems that (4.1) almost follows from these propositions, and I wonder if they may be helpful stepping stones in proving your (4.1).

266 (*error*) Replace ⌈ the same as Null ⌉ by: ⌈ the same as [Null] ⌉.

266 *ff.* (*rabbit*) Nothing prepares the reader for the magical appearance of $(\trianglelefteq)$. Was it hidden in the top hat, or lowered from the ceiling like a *deus ex machina*? It seems to me that its invention is driven by the need to have an ordering on spines that is compatible (when applicable) with the original one, and has a monotonicity property allowing the selection to be replaced by *last*. These needs basically force the definition of $(\trianglelefteq)$, in view of such equalities as $size\ (\mathtt{Fork}\ t\ u) = 2 + size\ t + size\ u$.

Is the detour via $(\trianglelefteq)$ really necessary? It may be better to observe that the last element of the list of builds always has the most sizeable head of the bunch, and then develop the machinery needed to prove this. It is possible to define a partial order on trees respecting their print sizes, where Null is at the bottom, and a fork dominates another fork when each of its children dominates the corresponding other child. Then the result of applying $map\ head\ \cdot\ map\ build\ \cdot\ inits$ is sorted. There are nice properties relating $build\ (xs +\!\!+ [x])$ to $build\ xs$, which can be proved by induction on xs: either $build\ (xs +\!\!+ [x])$ extends $build\ xs$ by snoccing a Null, or $build\ (xs +\!\!+ [x])$ has the same length, but, if not equal to $build\ xs$, differs in precisely one element, which is larger in the tree order. This should suffice to establish the most-sizeable head property. Perhaps the proposition "$head\ (build\ (xs +\!\!+ [x]))$ dominates $head\ (build\ xs)$" can also be

proved relatively easy by induction. Note: I have not checked in any detail whether this hopefully shorter path is actually feasible (and, if it is, whether it is actually simpler).

267 (*clarity, terminology*) Replace ⌈ ordering ⌉ by: ⌈ partial order ⌉.

268–269 (*clarity*) The functional programmer is likely to read this as a definition by cases, which, however fails. Consider $(map\ build\ \cdot\ inits)$ `")"`, which equals $[build\ \texttt{""}, build\ \texttt{")"}] = [\,[\mathsf{Null}], [\mathsf{Null}, \mathsf{Null}]\,]$. The use of $max_{\trianglelefteq}$ requires comparing $[\mathsf{Null}, \mathsf{Null}]$ with $[\mathsf{Null}]$. Well,

$$\begin{array}{rl} & [\mathsf{Null}, \mathsf{Null}] \trianglelefteq [\mathsf{Null}] \\ \equiv & \mathsf{Null} : [\mathsf{Null}] \trianglelefteq \mathsf{Null} : [\,] \\ \equiv & [\mathsf{Null}] \trianglelefteq [\,] \\ \equiv & ??? \end{array}$$

Neither of the two clauses applies. (The problem does not occur if all comparisons are between a pair with on the left an earlier element of $(map\, build \cdot inits)\ xs$ than on the right, but this is not made explicit – and there is, nevertheless, a proof obligation that the situation cannot occur; that $max_{\trianglelefteq}$ will never see $[\ldots, [\mathsf{Null}, \mathsf{Null}], [\mathsf{Null}], \ldots].$) The falsehood of $[\mathsf{Null}, \mathsf{Null}] \trianglelefteq [\mathsf{Null}]$ is implied by the antisymmetry of partial orders, but for the sake of clarity it is better to add a clause that covers the case. Replace

$$[\,] \trianglelefteq us \ \wedge$$

by:

$$\begin{array}{rcl} [\,] \trianglelefteq us & \equiv & \mathsf{true} \\ (t : ts) \trianglelefteq [\,] & \equiv & \mathsf{false} \end{array}$$

.

273–275 (*handwaving*) This skips a few steps. Why "must" $[t]$ be in the set of spines too? How does this follow, precisely, from $t : ts$ being maximal (not "largest") under $(\trianglelefteq)$?

277 (*error*) Replace ⌈ *bstep* is monotonic ⌉ by: ⌈ *bstep* x is monotonic ⌉.

302 (*proof presentation*) Include a hint here (⌈= {free theorem}⌉?) why the step is valid.

339 (*convention*) Replace ⌈ LNCS, no. 1816 ⌉ by: ⌈ LNCS 1816 ⌉. This is the conventional way of referring to volumes in the LNCS series. At the very least, replace ⌈no.⌉ by ⌈vol.⌉.

341 (*convention*) Replace ⌈ *(Special Issue for Mathematics of Program Construction)* ⌉ by: ⌈ (Special Issue: Mathematics of Program Construction (MPC 2002)) ⌉.

362 (*typography*) Replace ⌈ $xs=$ ⌉ by: ⌈ $xs\ \ =$ ⌉.

366 (*equality*) There is no reason to use ":=" here; this is not an assignment, definition or substitution. Replace $\lceil := \rceil$ by: $\lceil = \rceil$.

372 (*proof presentation*) You should include a validity hint here, referring to a separate proposition that presents the (not entirely trivial) relationship between $(stepsM\ ys \lll parseS)\ x$ and $parseS\ (ys +\!\!+ [x])$.