JFP **31**, e31, 10 pages, 2021.
doi:10.1017/S0956796821000253

FUNCTIONAL PEARL

Longest segment of balanced parentheses: an exercise in program inversion in a segment problem

SHIN-CHENG MU
Institute of Information Science, Academia Sinica, Taipei, Taiwan
(*e-mail:* scm@iis.sinica.edu.tw)

TSUNG-JU CHIANG
Institute of Information Science, Academia Sinica, Taipei, Taiwan
(*e-mail:* mail@tsung-ju.org)

Abstract

Given a string of parentheses, the task is to find the longest consecutive segment that is balanced, in linear time. We find this problem interesting because it involves a combination of techniques: the usual approach for solving segment problems and a theorem for constructing the inverse of a function—through which we derive an instance of shift-reduce parsing.

1 Introduction

Given a string of parentheses, the task is to find a longest consecutive segment that is balanced. For example, for input `"))(()())())()("` the output should be `"(()())()"`. We also consider a reduced version of the problem in which we return only the length of the segment. While there is no direct application of this problem,[1] the authors find it interesting because it involves two techniques. Firstly, derivation for such *optimal segment* problems (those whose goal is to compute a segment of a list that is optimal up to certain criteria) usually follows a certain pattern (e.g. Bird 1987; Zantema 1992; Gibbons 1997). We would like to see how well that works for this case. Secondly, at one point, we will need to construct the right inverse of a function. It will turn out that we will discover an instance of shift-reduce parsing.

Specification Balanced parentheses can be captured by a number of grammars, for example, $\mathsf{S} \rightarrow \epsilon \mid (\mathsf{S}) \mid \mathsf{SS}$, or $\mathsf{S} \rightarrow \mathsf{T}^*$ and $\mathsf{T} \rightarrow (\mathsf{S})$. After trying some of them, the authors decided on

$$\mathsf{S} \rightarrow \epsilon \mid (\mathsf{S})\,\mathsf{S},$$

[1] However, the length-only version was possibly used as an interview problem collected in, for example, https://leetcode.com/problems/longest-valid-parentheses/.

because it is unambiguous and the most concise. Other grammars have worked too, albeit leading to lengthier algorithms. The parse tree of the chosen grammar can be represented in Haskell as below, with a function *pr* specifying how a tree is printed:

$$\mathbf{data}\ \mathsf{Tree} = \mathsf{Nul} \mid \mathsf{Bin}\ \mathsf{Tree}\ \mathsf{Tree},$$

$$\begin{aligned} &pr :: \mathsf{Tree} \to \mathsf{String} \\ &pr\ \mathsf{Nul} \quad\quad = \texttt{""} \\ &pr\ (\mathsf{Bin}\ t\ u) = \texttt{"("} +\!\!+ pr\ t +\!\!+ \texttt{")"} +\!\!+ pr\ u. \end{aligned}$$

For example, letting $t_1 = \mathsf{Bin\ Nul\ Nul}$ and $t_2 = \mathsf{Bin\ Nul\ (Bin\ Nul\ Nul)}$, we have $pr\ t_1 =$ "()", $pr\ t_2 =$ "()()" and $pr\ (\mathsf{Bin}\ t_2\ t_1) =$ "(()())()" (parentheses are coloured to aid the reader).

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 $parse :: \mathsf{String} \to \mathsf{Maybe\ Tree}$ builds a parse tree—*parse xs* should return Just *t* such that $pr\ t = xs$ if *xs* is balanced and return Nothing otherwise. While this defines *parse* already, a direct definition of *parse* will be presented in Section 4.

The problem can then be specified as below, where *lbs* stands for "longest balanced segment (of parentheses)":

$$\begin{aligned} &lbs :: \mathsf{String} \to \mathsf{Tree} \\ &lbs = maxBy\ size \cdot filtJust \cdot map\ parse \cdot segments, \end{aligned}$$

$$\begin{aligned} &segments = concat \cdot map\ inits \cdot tails, \\ &filtJust\ ts = [t \mid \mathsf{Just}\ t \leftarrow ts], \\ &size\ t = length\ (pr\ t). \end{aligned}$$

The function $segments :: [a] \to [[a]]$ returns all segments of a list, with $inits, tails :: [a] \to [[a]]$, respectively, computing all prefixes and suffixes of their input lists. The result of *map parse* is passed to $filtJust :: [\mathsf{Maybe}\ a] \to [a]$, which collects only those elements wrapped by Just. For example, $filtJust\ [\mathsf{Just}\ 1, \mathsf{Nothing}, \mathsf{Just}\ 2] = [1, 2]$.[2] For this problem *filtJust* always returns a non-empty list, because the empty string, which is a member of *segments xs* for any *xs*, can always be parsed to Just Nul. Given $f :: a \to b$ where b is a type that is ordered, $maxBy\ f :: [a] \to a$ picks a maximum element from the input. Finally, *size t* computes the length of *pr t*.

The length-only problem can be specified by $lbsl = size \cdot lbs$.

2 The prefix–suffix decomposition

It is known that many optimal segment problems can be solved by following a fixed pattern (Bird, 1987; Zantema, 1992; Gibbons, 1997), which we refer to as *prefix–suffix decomposition*. In the first step, finding an optimal segment is factored into finding, for each suffix, an optimal prefix. For our problem, the calculation goes

[2] *filtJust* is called *catMaybes* in the standard Haskell libraries. The authors think the name *filtJust* is more informative.

https://doi.org/10.1017/S0956796821000253 Published online by Cambridge University Press

$$
\begin{array}{ll}
& \mathit{maxBy\ size} \cdot \mathit{filtJust} \cdot \mathit{map\ parse} \cdot \mathit{segments} \\
= & \{ \text{ definition of } \mathit{segments} \ \} \\
& \mathit{maxBy\ size} \cdot \mathit{filtJust} \cdot \mathit{map\ parse} \cdot \mathit{concat} \cdot \mathit{map\ inits} \cdot \mathit{tails} \\
= & \{ \text{ since } \mathit{map\ f} \cdot \mathit{concat} = \mathit{concat} \cdot \mathit{map}\ (\mathit{map\ f}),\ \mathit{map} \text{ fusion } \} \\
& \mathit{maxBy\ size} \cdot \mathit{filtJust} \cdot \mathit{concat} \cdot \mathit{map}\ (\mathit{map\ parse} \cdot \mathit{inits}) \cdot \mathit{tails} \\
= & \{ \text{ since } \mathit{filtJust} \cdot \mathit{concat} = \mathit{concat} \cdot \mathit{map\ filtJust} \ \} \\
& \mathit{maxBy\ size} \cdot \mathit{concat} \cdot \mathit{map}\ (\mathit{filtJust} \cdot \mathit{map\ parse} \cdot \mathit{inits}) \cdot \mathit{tails} \\
= & \{ \text{ since } \mathit{maxBy\ f} \cdot \mathit{concat} = \mathit{maxBy\ f} \cdot \mathit{map}\ (\mathit{maxBy\ f}) \ \} \\
& \mathit{maxBy\ size} \cdot \mathit{map}\ (\mathit{maxBy\ size} \cdot \mathit{filtJust} \cdot \mathit{map\ parse} \cdot \mathit{inits}) \cdot \mathit{tails}.
\end{array}
$$

For each suffix returned by *tails*, the program above computes its longest *prefix* of balanced parentheses by $\mathit{maxBy\ size} \cdot \mathit{filtJust} \cdot \mathit{map\ parse} \cdot \mathit{inits}$. We abbreviate the latter to *lbp* (for "longest balanced prefix").

Generating every suffix and computing *lbp* for each of them is rather costly. The next step is to try to apply the following *scan lemma*, which says that if a function f can be expressed as right fold, there is a more efficient algorithm to compute $\mathit{map\ f} \cdot \mathit{tails}$:

Lemma 2.1. $\mathit{map}\ (\mathit{foldr}\ (\oplus)\ e) \cdot \mathit{tails} = \mathit{scanr}\ (\oplus)\ e$, where

$$
\begin{array}{l}
\mathit{scanr} :: (a \to b \to b) \to b \to [a] \to [b] \\
\mathit{scanr}\ (\oplus)\ e\ [\,] \qquad = [e] \\
\mathit{scanr}\ (\oplus)\ e\ (x : xs) = \mathbf{let}\ (y : ys) = \mathit{scanr}\ (\oplus)\ e\ xs\ \mathbf{in}\ (x \oplus y) : y : ys.
\end{array}
$$

If *lbp* can be written in the form $\mathit{foldr}\ (\oplus)\ e$, we do not need to compute *lbp* of each suffix from scratch; each optimal prefix can be computed, in *scanr*, from the previous optimal prefix by $(\oplus)$. If $(\oplus)$ is a constant-time operation, we get a linear-time algorithm.

The next challenge is therefore to express *lbp* as a right fold. Since *inits* can be expressed as a right fold—$\mathit{inits} = \mathit{foldr}\ (\lambda x\ xss \to [\,] : \mathit{map}\ (x{:})\ xss)\ [[\,]]$, a reasonable attempt is to fuse $\mathit{maxBy\ size} \cdot \mathit{filtJust} \cdot \mathit{map\ parse}$ with *inits*, to form a single *foldr*. Recall the *foldr*-fusion theorem:

Theorem 2.2 (*foldr*-fusion). $h \cdot \mathit{foldr}\ f\ e = \mathit{foldr}\ g\ (h\ e)$ if $h\ (f\ x\ y) = g\ x\ (h\ y)$.

The antecedent $h\ (f\ x\ y) = g\ x\ (h\ y)$ will be referred to as the *fusion condition*. To fuse *map parse* and *inits* using Theorem 2.2, we calculate from the LHS of the fusion condition (with $h = \mathit{map\ parse}$ and $f = (\lambda x\ xss \to [\,] : \mathit{map}\ (x{:})\ xss)$):

$$
\begin{array}{ll}
& \mathit{map\ parse}\ ([\,] : \mathit{map}\ (x{:})\ xss) \\
= & \{ \text{ since } \mathit{parse}\ [\,] = \mathsf{Just\ Nul} \ \} \\
& \mathsf{Just\ Nul} : \mathit{map}\ (\mathit{parse} \cdot (x{:}))\ xss \\
= & \{ \text{ wish, for some } g' \ \} \\
& \mathsf{Just\ Nul} : g'\ x\ (\mathit{map\ parse\ xss}) \\
= & \{ \text{ let } g\ x\ ts = \mathsf{Just\ Nul} : g\ x\ ts \ \} \\
& g\ x\ (\mathit{map\ parse\ xss}).
\end{array}
$$

We can construct g if (and only if) there is a function g' such that $g'\ x\ (\mathit{map\ parse\ xss}) = \mathit{map}\ (\mathit{parse} \cdot (x{:}))\ xss$. Is that possible?

It is not hard to see that the answer is no. Consider $xss = [\texttt{")"}, \texttt{")()"}]$ and $x = \texttt{'('}$. Since both strings in *xss* are not balanced, $\mathit{map\ parse\ xss}$ gives us $[\mathsf{Nothing}, \mathsf{Nothing}]$.

https://doi.org/10.1017/S0956796821000253 Published online by Cambridge University Press

However, $map\,(x:)\,xss = [\texttt{"()"}, \texttt{"()()"}]$, a list of balanced strings. Therefore, g' has to produce something from nothing—an impossible task. We have to generalise our problem such that g' receives inputs that are more informative.

3 Partially balanced strings

A string of parentheses is said to be *left-partially balanced* if it may possibly be balanced by adding zero or more parentheses to the left. For example, $xs = \texttt{"(())()))()"}$ is left-partially balanced because $\texttt{"(("} +\!\!+ xs$ is balanced—again we use colouring to help the reader parsing the string. Note that $\texttt{"(()("} +\!\!+ xs$ is also balanced. For a counter example, the string $ys = \texttt{"()(()"}$ is not left-partially balanced—due to the unmatched '(' in the middle of ys, there is no zs such that $zs +\!\!+ ys$ can be balanced.

While parsing a *fully balanced* string cannot be expressed as a right fold, it is possible to parse *left-partially balanced* strings using a right fold. In this section, we consider what data structure such a string should be parsed to. We discuss how to how to parse it in the next section.

A left-partially balanced string can always be uniquely factored into a sequence of fully balanced substrings, separated by one or more right parentheses. For example, xs can be factored into two balanced substrings, "(())()" and "()", separated by "))". One of the possible ways to represent such a string is by a list of trees—a Forest, where the trees are supposed to be separated by a ')'. That is, such a forest can be printed by:

$$
\begin{aligned}
&\textbf{type}\ \mathsf{Forest} = [\mathsf{Tree}], \qquad\qquad \{-\ \text{non-empty}\ -\}\\
&prF :: \mathsf{Forest} \to \mathsf{String}\\
&prF\,[t] \quad\ = pr\,t\\
&prF\,(t:ts) = pr\,t +\!\!+ \texttt{")"} +\!\!+ prF\,ts.
\end{aligned}
$$

For example, $xs = \texttt{"(())()))()"}$ can be represented by a forest containing three trees:

$$ts = [\mathsf{Bin}\,(\mathsf{Bin}\,\mathsf{Nul}\,\mathsf{Nul})\,(\mathsf{Bin}\,\mathsf{Nul}\,\mathsf{Nul}),\quad \mathsf{Nul},\quad \mathsf{Bin}\,\mathsf{Nul}\,\mathsf{Nul}],$$

where $\mathsf{Bin}\,(\mathsf{Bin}\,\mathsf{Nul}\,\mathsf{Nul})\,(\mathsf{Bin}\,\mathsf{Nul}\,\mathsf{Nul})$ prints to "(())()", $\mathsf{Bin}\,\mathsf{Nul}\,\mathsf{Nul}$ prints to "()", and there is a Nul between them due to the consecutive right parentheses "))" in xs (Nul itself prints to ""). One can verify that $prF\,ts = xs$ indeed. Note that we let the type Forest be *non-empty* lists of trees.[3] The empty string can be represented by [Nul], since $prF\,[\mathsf{Nul}] = pr\,\mathsf{Nul} = \texttt{""}$.

The aim now is to construct the right inverse of prF, such that a left-partially balanced string can be parsed using a right fold.

4 Parsing partially balanced strings

Given a function $f :: b \to t$, the converse-of-a-function theorem (Bird & de Moor, 1997; de Moor & Gibbons, 2000) constructs the relational converse—a generalised notion of inverse—of f. The converse is given as a relational fold whose input type is t, which can

[3] We can let the non-emptiness be more explicit by letting $\mathsf{Forest} = (\mathsf{Tree}, [\mathsf{Tree}])$. Presentation-wise, both representations have their pros and cons, and we eventually decided on using a list.

https://doi.org/10.1017/S0956796821000253 Published online by Cambridge University Press

be any inductively defined datatype with a polynomial base functor. We specialise the general theorem to our needs: we use it to construct only functions, not relations, and only for the case where t is a list type.

Theorem 4.1. Given $f :: b \to [a]$, if we have $base :: b$ and $step :: a \to b \to b$ satisfying:

$$f\ base = [\,] \quad \wedge$$
$$f\ (step\ x\ t) = x : f\ t,$$

then $f^{-1} = foldr\ step\ base$ is a partial right inverse of f. That is, we have $f\ (f^{-1}\ xs) = xs$ for all xs in the range of f.

While the general version of the theorem is not trivial to prove, the version above, specialised to functions and lists, can be verified by an easy induction on the input list.

Recall that we wish to construct the right inverse of prF using Theorem 4.1. It will be easier if we first construct a new definition of prF, one that is inductive, does not use $(+\!\!+)$ and does not rely on pr. For a base case, $prF\ [\mathsf{Nul}] = \texttt{""}$. It is also immediate that $prF\ (\mathsf{Nul} : ts) = \texttt{')'} : prF\ ts$. When the list contains more than one tree and the first tree is not Nul, we calculate

$$\begin{array}{ll} & prF\ (\mathsf{Bin}\ t\ u : ts) \\ = & \quad \{\text{ definitions of } pr \text{ and } prF\ \} \\ & \texttt{"("} +\!\!+ pr\ t +\!\!+ \texttt{")"} +\!\!+ pr\ u +\!\!+ \texttt{")"} +\!\!+ prF\ ts \\ = & \quad \{\text{ definition of } prF\ \} \\ & \texttt{'('} : prF\ (t : u : ts). \end{array}$$

We have thus derived the following new definition of prF:

$$\begin{array}{ll} prF\ [\mathsf{Nul}] & = \texttt{""} \\ prF\ (\mathsf{Nul} : ts) & = \texttt{')'} : prF\ ts \\ prF\ (\mathsf{Bin}\ t\ u : ts) & = \texttt{'('} : prF\ (t : u : ts). \end{array}$$

We are now ready to invert prF by Theorem 4.1, which amounts to finding $base$ and $step$ such that $prF\ base = \texttt{""}$ and $prF\ (step\ x\ ts) = x : prF\ ts$ for $x = \texttt{'('}$ or $\texttt{')'}$. With the inductive definition of prF in mind, we pick $base = [\mathsf{Nul}]$, and the following $step$ meets the requirement:

$$\begin{array}{ll} step\ \texttt{')'}\ ts & = \mathsf{Nul} : ts \\ step\ \texttt{'('}\ (t : u : ts) & = \mathsf{Bin}\ t\ u : ts. \end{array}$$

We have thus constructed $prF^{-1} = foldr\ step\ [\mathsf{Nul}]$. If we expand the definitions, we have

$$\begin{array}{ll} prF^{-1} :: \mathsf{String} \to \mathsf{Forest} & \\ prF^{-1}\ \texttt{""} & = [\mathsf{Nul}] \\ prF^{-1}\ (\texttt{')'} : xs) & = \mathsf{Nul} : prF^{-1}\ xs \\ prF^{-1}\ (\texttt{'('} : xs) & = \mathbf{case}\ prF^{-1}\ xs\ \mathbf{of}\ (t : u : ts) \to \mathsf{Bin}\ t\ u : ts, \end{array}$$

which is pleasingly symmetrical to the inductive definition of prF.

For an operational explanation, a right parenthesis `')'` indicates starting a new tree, thus we start freshly with a Nul; a left parenthesis `'('` ought to be the leftmost symbol of some `"(t)u"`, thus we wrap the two most recent siblings into one tree. When there are

https://doi.org/10.1017/S0956796821000253 Published online by Cambridge University Press

no such two siblings (i.e. $prF^{-1}\ xs$ in the **case** expression evaluates to a singleton list), the input `'('` $: xs$ is not a left-partially balanced string—`'('` appears too early, and the result is undefined.

Readers may have noticed the similarity to shift-reduce parsing, in which, after reading a symbol we either "shift" the symbol by pushing it onto a stack, or "reduce" the symbol against a top segment of the stack. Here, the forest is the stack. The input is processed right-to-left, as opposed to left-to-right, which is more common when talking about parsing. We shall discuss this issue further in Section 7.

We could proceed to work with prF^{-1} for the rest of this pearl but, for clarity, we prefer to make the partiality explicit. Let *parseF* be the monadified version of prF^{-1}, given by:

```
parseF :: String → Maybe Forest
parseF ""       = Just [Nul]
parseF (x : xs) = parseF xs >>= stepM x,
  where stepM ')' ts          = Just (Nul : ts)
        stepM '(' [t]         = Nothing
        stepM '(' (t : u : ts) = Just (Bin t u : ts),
```

where *stepM* is monadified *step*—for the case $[t]$ missing in *step* we return Nothing.

To relate *parseF* to *parse*, notice that $prF\ [t] = pr\ t$. We therefore have

```
parse :: String → Maybe Tree
parse = unwrapM <=< parseF,

unwrapM [t] = Just t
unwrapM _   = Nothing.
```

where $(\lessdot\!\!<) :: (b \rightarrow \mathsf{M}\ c) \rightarrow (a \rightarrow \mathsf{M}\ b) \rightarrow (a \rightarrow \mathsf{M}\ c)$ is (reversed) Kleisli composition. That is, *parse* calls *parseF* and declares success only when the input can be parsed into a single tree.

5 Longest balanced prefix in a fold

Recall our objective at the close of Section 2: to compute $lbp = maxBy\ size \cdot filtJust \cdot map\ parse \cdot inits$ in a right fold in order to obtain a faster algorithm using the scan lemma. Now that we have $parse = unwrapM \lessdot\!\!< parseF$ where *parseF* is a right fold, and we perform some initial calculation whose purpose is to factor the post-processing *unwrapM* out of the main computation:

$$
\begin{aligned}
& maxBy\ size \cdot filtJust \cdot map\ parse \cdot inits \\
= & \quad \{ \text{ since } parse = unwrapM \lessdot\!\!< parseF \ \} \\
& maxBy\ size \cdot filtJust \cdot map\ (unwrapM \lessdot\!\!< parseF) \cdot inits \\
= & \quad \{ \text{ since } (f \lessdot\!\!< g)\ x = f =\!\!<\!\!< g\ x, \text{ map fusion (backwards) } \} \\
& maxBy\ size \cdot filtJust \cdot map\ (unwrapM =\!\!<\!\!<) \cdot map\ parseF \cdot inits \\
= & \quad \{ \text{ since } filtJust \cdot map\ (unwrapM =\!\!<\!\!<) = map\ unwrap \cdot filtJust, \text{ see below } \} \\
& maxBy\ size \cdot map\ unwrap \cdot filtJust \cdot map\ parseF \cdot inits \\
= & \quad \{ \text{ since } maxBy\ f \cdot map\ g = g \cdot maxBy\ (f \cdot g) \ \} \\
& unwrap \cdot maxBy\ (size \cdot unwrap) \cdot filtJust \cdot map\ parseF \cdot inits.
\end{aligned}
$$

https://doi.org/10.1017/S0956796821000253 Published online by Cambridge University Press

In the penultimate step $(unwrapM \mathrel{=\!\!<\!\!<})$ is moved leftwards past *filtJust* and becomes $unwrap :: \mathsf{Forest} \rightarrow \mathsf{Tree}$, defined by:

$$unwrap\,[t] = t$$
$$unwrap\ \_ \ \ = \mathsf{Nul}.$$

Recall that $inits = foldr\,(\lambda x\,xss \rightarrow [\,] : map\,(x:)\,xss)\,[[\,]]$. The aim now is to fuse $map\ parseF$, *filtJust* and $maxBy\,(size \cdot unwrap)$ with *inits*.

By Theorem 2.2, to fuse $map\ parseF$ with *inits*, we need to construct g that meets the fusion condition:

$$map\ parseF\ ([\,] : map\,(x:)\,xss) = g\,x\,(map\ parseF\ xss).$$

Now that we know that *parseF* is a fold, the calculation goes

$$\begin{aligned}
&\ \ map\ parseF\ ([\,] : map\,(x:)\,xss)\\
=&\ \ \{\ \text{definitions of } map \text{ and } parseF\ \}\\
&\ \ \mathsf{Just}\,[\mathsf{Nul}] : map\,(parseF \cdot (x:))\,xss\\
=&\ \ \{\ \text{the } foldr \text{ definition of } parseF\ \}\\
&\ \ \mathsf{Just}\,[\mathsf{Nul}] : map\,(\lambda ts \rightarrow parseF\ ts \gg\!= stepM\ x)\,xss\\
=&\ \ \{\ map\text{-fusion (backwards)}\ \}\\
&\ \ \mathsf{Just}\,[\mathsf{Nul}] : map\,(\gg\!= stepM\ x)\,(map\ parseF\ xss).
\end{aligned}$$

Therefore, we have

$$\begin{aligned}
&map\ parseF \cdot inits :: \mathsf{String} \rightarrow [\mathsf{Maybe\ Forest}]\\
&map\ parseF \cdot inits =\\
&\quad foldr\,(\lambda x\ tss \rightarrow \mathsf{Just}\,[\mathsf{Nul}] : map\,(\gg\!= stepM\ x)\ tss)\,[\mathsf{Just}\,[\mathsf{Nul}]].
\end{aligned}$$

Next, we fuse *filtJust* with $map\ parseF \cdot inits$ by Theorem 2.2. After some calculations, we get

$$\begin{aligned}
&filtJust \cdot map\ parseF \cdot inits :: \mathsf{String} \rightarrow [\mathsf{Forest}]\\
&filtJust \cdot map\ parseF \cdot inits = foldr\,(\lambda x\ tss \rightarrow [\mathsf{Nul}] : extend\ x\ tss)\,[[\mathsf{Nul}]],\\
&\quad \mathbf{where}\ extend\ \texttt{')'}\ tts = map\,(\mathsf{Nul}:)\ tts\\
&\qquad\qquad\ \ extend\ \texttt{'('}\ tts = [(\mathsf{Bin}\ t\ u : ts) \mid (t : u : ts) \leftarrow tts].
\end{aligned}$$

After the fusion, we need not keep the Nothing entries in the fold; the computation returns a collection of forests. If the next character is `')'`, we append Nul to every forest. If the next entry is `'('`, we choose those forests having at least two trees and combine them—the list comprehension keeps only the forests that match the pattern $(t : u : ts)$ and throws away those do not. Note that [Nul], to which the empty string is parsed, is always added to the collection of forests.

To think about how to deal with $unwrap \cdot maxBy\,(size \cdot unwrap)$, we consider an example. Figure 1 shows the results of $map\ parseF$ and *filtJust* for prefixes of `"())()("`, where Just, Nul and Bin are, respectively, abbreviated to J, N and B. The function $maxBy\,(size \cdot unwrap)$ chooses between [N] and [B N N], the two parses resulting in single trees, and returns [B N N]. However, notice that B N N is also the head of [B N N, B N N], the last forest returned by *filtJust*. In general, the largest singleton parse tree will also present in the head of the last forest returned by $filtJust \cdot map\ parseF \cdot inits$. One can

https://doi.org/10.1017/S0956796821000253 Published online by Cambridge University Press

inits	*map parseF*	*filtJust*
""	J [N]	[N]
"("	Nothing	
"()"	J [B N N]	[B N N]
"())"	J [B N N, N]	[B N N, N]
"())("	Nothing	
"())()"	J [B N N, B N N]	[B N N, B N N]
"())()("	Nothing	

Fig. 1: Results of *parseF* and *filtJust* for prefixes of "())()(".

intuitively see why: if we print them both, the former is a prefix of the latter. Therefore, $unwrap \cdot maxBy\,(size \cdot unwrap)$ can be replaced by $head \cdot last$.

To fuse *last* with $filtJust \cdot map\ parseF \cdot inits$ by Theorem 2.2, we need to construct a function *step* that satisfies the fusion condition:

$$last\,([\mathsf{Nul}] : extend\ x\ tss) = step\ x\,(last\ tss),$$

where *tss* is a non-empty list of forests. The case when $x =$ ')' is easy—choosing $step$ ')' $ts = \mathsf{Nul} : ts$ will do the job. For the case when $x =$ '(' we need to analyse the result of *last tss* and use the property that forests in *tss* are ordered in ascending lengths.

(a) If $last\ tss = [t]$, a forest having only one tree, there are no forest in *tss* that contains two or more trees. Therefore, *extend* '(' *tss* returns an empty list, and *last* ([Nul] : *extend* '(' *tss*) = [Nul].
(b) Otherwise, *extend* '(' *tss* would not be empty, and $last\,([\mathsf{Nul}] : extend\ x\ tss) = last\,(extend\ x\ tss)$. We may then combine the first two trees, as *extend* would do.

In summary, we have

$$
\begin{aligned}
&last \cdot filtJust \cdot map\ parseF \cdot inits :: \mathsf{String} \to \mathsf{Forest}\\
&last \cdot filtJust \cdot map\ parseF \cdot inits = foldr\ step\ [\mathsf{Nul}],\\
&\quad \textbf{where}\ step\ \text{')'}\ ts \quad = \mathsf{Nul} : ts\\
&\qquad\quad step\ \text{'('}\ [t] \quad = [\mathsf{Nul}]\\
&\qquad\quad step\ \text{'('}\ (t : u : ts) = \mathsf{Bin}\ t\ u : ts,
\end{aligned}
$$

which is now a total function on strings of parentheses.

The function derived above turns out to be prF^{-1} with one additional case (*step* '(' $[t] = [\mathsf{Nul}]$). What we have done in this section can be seen as justifying this extra case (which is a result of case (1) in the fusion of *last*), which is not as trivial as one might think.

6 Wrapping up

We can finally resume the main derivation in Section 2:

$$
\begin{aligned}
&\quad maxBy\ size \cdot map\,(maxBy\ size \cdot filtJust \cdot map\ parse \cdot inits) \cdot tails\\
&= \quad \{\ \text{Section 5: } lbp = head \cdot foldr\ step\ [\mathsf{Nul}]\ \}\\
&\quad maxBy\ size \cdot map\,(head \cdot foldr\ step\ [\mathsf{Nul}]) \cdot tails\\
&= \quad \{\ map\text{-fusion reversed, Lemma 2.1}\ \}\\
&\quad maxBy\ size \cdot map\ head \cdot scanr\ step\ [\mathsf{Nul}].
\end{aligned}
$$

https://doi.org/10.1017/S0956796821000253 Published online by Cambridge University Press

input size (M)	1	2	4	6	8	10
user time (sec.)	0.52	1.25	2.38	3.20	4.74	5.50

Fig. 2: Measured running time for some input sizes.

We have therefore derived:

$$lbs :: \mathsf{String} \rightarrow \mathsf{Tree}$$
$$lbs = maxBy\ size \cdot map\ head \cdot scanr\ step\ [\mathsf{Nul}],$$

where *step* is as defined in the end of Section 5. To avoid recomputing the sizes in *maxBy size*, we can annotate each tree by its size: letting $\mathsf{Forest} = [(\mathsf{Tree}, \mathsf{Int})]$, resulting in an algorithm that runs in linear time:

$$\begin{aligned}
&lbs :: \mathsf{String} \rightarrow \mathsf{Tree}\\
&lbs = fst \cdot maxBy\ snd \cdot map\ head \cdot scanr\ step\ [(\mathsf{Nul}, 0)],\\
&\quad \textbf{where}\ step\ \text{'})\text{'}\ ts \quad = (\mathsf{Nul}, 0) : ts\\
&\qquad\quad step\ \text{'}(\text{'}\ [t] = [(\mathsf{Nul}, 0)]\\
&\qquad\quad step\ \text{'}(\text{'}\ ((t, m) : (u, n) : ts) = (\mathsf{Bin}\ t\ u, 2 + m + n) : ts.
\end{aligned}$$

Finally, the size-only version can be obtained by fusing *size* into *lbs*. It turns out that we do not need to keep the actual trees, but only their sizes—$\mathsf{Forest} = [\mathsf{Int}]$:

$$\begin{aligned}
&lbsl :: \mathsf{String} \rightarrow \mathsf{Int}\\
&lbsl = maximum \cdot map\ head \cdot scanr\ step\ [0],\\
&\quad \textbf{where}\ step\ \text{'})\text{'}\ ts \quad = 0 : ts\\
&\qquad\quad step\ \text{'}(\text{'}\ [t] = [0]\\
&\qquad\quad step\ \text{'}(\text{'}\ (m : n : ts) = (2 + m + n) : ts.
\end{aligned}$$

We ran some simple experiments to measure the efficiency of the algorithm. The test machine was a laptop computer with a Apple M1 chip (8 core, 3.2GHz) and 16GB RAM. We ran *lbs* on randomly generated inputs containing 1, 2, 4, 6, 8 and 10 million parentheses and measured the user times. The results, shown in Figure 2, confirmed the linear-time behaviour.

7 Conclusions and discussions

So we have derived a linear-time algorithm for solving the problem. We find it an interesting journey because it relates two techniques: prefix–suffix decomposition for solving segment problems and the converse-of-a-function theorem for program inversion.

In Section 3, we generalised from trees to forests. Generalisations are common when applying the converse-of-a-function theorem. It was observed that the trees in a forest are those along the left spine of the final tree; therefore, such a generalisation is referred to as switching to a "spine representation" (Mu & Bird, 2003).

What we derived in Sections 4 and 5 is a compacted form of shift-reduce parsing, where the input is processed right to left. The forest serves as the stack, but we do not need to push the parser state to the stack, as is done in shift-reduce parsing. If we were to process the input in the more conventional left-to-right order, the corresponding grammar

https://doi.org/10.1017/S0956796821000253 Published online by Cambridge University Press

would be $\mathsf{S} \rightarrow \epsilon \mid \mathsf{S}\,(\mathsf{S})$. It is an SLR(1) grammar whose parse table contains five states. Our program is much simpler. A possible reason is that consecutive shifting and reducing are condensed into one step. It is likely that parsing SLR(1) languages can be done in a fold. The relationship between LR parsing and the converse-of-a-function theorem awaits further investigation.

There are certainly other ways to solve the problem. For example, one may interpret a '(' as a -1, and a ')' as a $+1$. A left-partially balanced string would be a list whose right-to-left running sum is never negative. One may then apply the method in Zantema (1992) to find the longest such prefix for each suffix. The result will be an algorithm that maintains the sum in a loop—an approach that might be more commonly adopted by imperative programmers. The problem can also be seen as an instance of *maximum-marking problems*—choosing elements in a data structure that meet a given criteria while maximising a cost function—to which methods of Sasano *et al.* (2001) can be applied.

Acknowledgements

The problem was suggested by Yi-Chia Chen. The authors would like to thank our colleagues in the PLFM group in IIS, Academia Sinica, in particular Hsiang-Shang 'Josh' Ko, Liang-Ting Chen and Ting-Yan Lai, for valuable discussions. Also thanks to Chung-Chieh Shan and Kim-Ee Yeoh for their advice on earlier drafts of this paper. We are grateful to the reviewers of previous versions of this article who gave detailed and constructive criticisms that helped a lot in improving this work.

Conflicts of interest

None.

References

Bird, R. S. (1987) An introduction to the theory of lists. In Broy, M. (ed), *Logic of Programming and Calculi of Discrete Design*. NATO ASI Series F, vol. 36. Springer, pp. 5–42.

Bird, R. S. & de Moor, O. (1997) *Algebra of Programming*. Prentice Hall.

de Moor, O. & Gibbons, J. (2000) Pointwise relational programming. In Rus, T. (ed), *Algebraic Methodology and Software Technology*. LNCS, vol. 1816. Springer, pp. 371–390.

Gibbons, J. (1997) Calculating functional programs. In Proceedings of ISRG/SERG Research Colloquium. Oxford Brookes University.

Mu, S.-C. & Bird, R. S. (2003) Theory and applications of inverting functions as folds. *Sci. Comput. Program. (Special Issue Math. Program Const.)* **51**, 87–116.

Sasano, I., Hu, Z. & Takeichi, M. (2001) Generation of efficient programs for solving maximum multi-marking problems. In ACM SIGPLAN Workshop on Semantics, Applications and Implementation of Program Generation (SAIG'01). LNCS, vol. 2196. Springer, pp. 72–91.

Zantema, H. (1992) Longest segment problems. *Sci. Comput. Program.* **18**(1), 39–66.