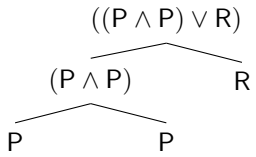
Peter Smith, *Introduction to Formal Logic* (CUP, 2nd edition)

Exercises 9: PL syntax (parse trees)

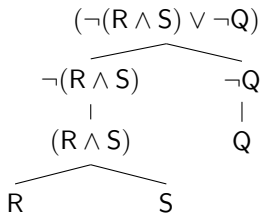
(a) *Show the following expressions are wffs of a PL language with suitable atoms by producing parse trees. Which is the main connective of each wff? What is the scope of each connective in (3) and of each disjunction in (4)? List all the subformulas of (5). Use alternative styles of brackets in (5) and (6) to make them more easily readable.*

(1) $((\mathsf{P} \wedge \mathsf{P}) \vee \mathsf{R})$



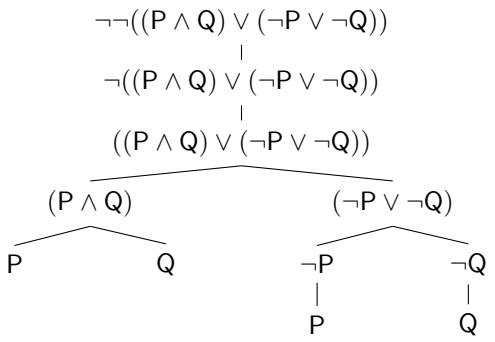
Main connective: $\vee$

(2) $(\neg(\mathsf{R} \wedge \mathsf{S}) \vee \neg\mathsf{Q})$



Main connective: $\vee$

(3) $\neg\neg((\mathsf{P} \wedge \mathsf{Q}) \vee (\neg\mathsf{P} \vee \neg\mathsf{Q}))$



Main connective: $\neg$

The scope of the initial $\neg$ is the whole wff.

The scope of the second $\neg$ is $\neg((\mathsf{P} \wedge \mathsf{Q}) \vee (\neg\mathsf{P} \vee \neg\mathsf{Q}))$.

The scope of $\wedge$ is $(\mathsf{P} \wedge \mathsf{Q})$.

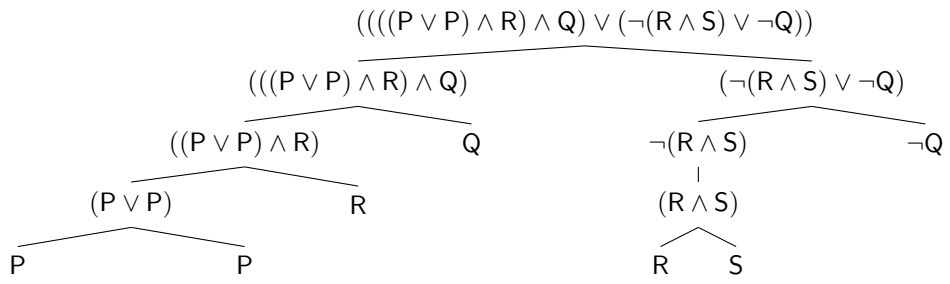
The scope of the first $\vee$ is $((\mathsf{P} \wedge \mathsf{Q}) \vee (\neg\mathsf{P} \vee \neg\mathsf{Q}))$.

The scope of the third $\neg$ is just $\neg\mathsf{P}$.

The scope of the second $\vee$ is $(\neg\mathsf{P} \vee \neg\mathsf{Q})$.

And the scope of the final $\neg$ is just $\neg\mathsf{Q}$.

(4) $((((P \vee P) \wedge R) \wedge Q) \vee (\neg(R \wedge S) \vee \neg Q))$



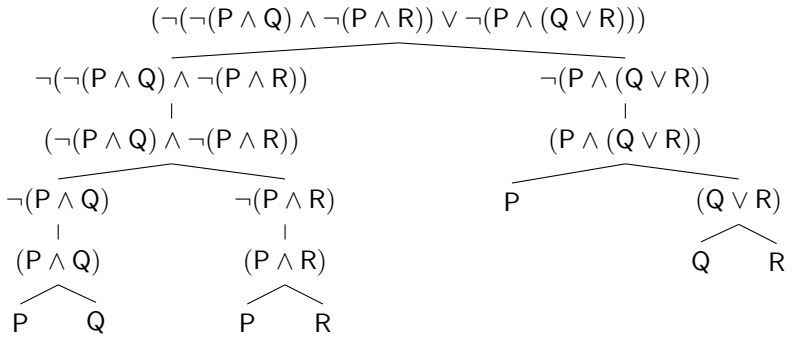
Main connective: $\vee$

The scope of the first $\vee$ is $(P \vee P)$.

The scope of the second $\vee$ is the whole wff.

The scope of the third $\vee$ is $(\neg(R \wedge S) \vee \neg Q)$.

(5) $(\neg(\neg(P \wedge Q) \wedge \neg(P \wedge R)) \vee \neg(P \wedge (Q \vee R)))$



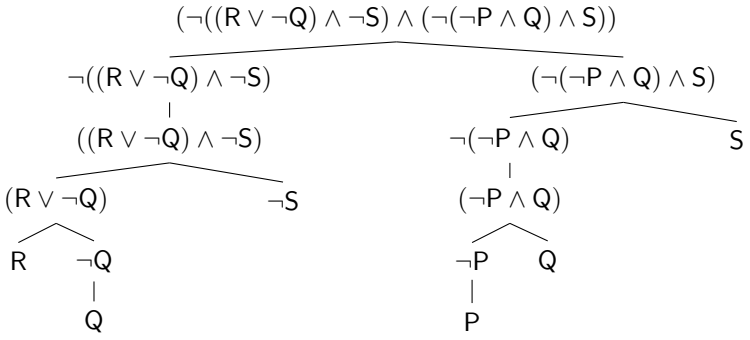
Main connective: $\vee$

Every wff that appears anywhere on the tree, including at the very top, counts as a subwff of the initial wff.

The wff can be made more readable by changing bracketing styles, e.g. like this:

$$[\neg\{\neg(P \wedge Q) \wedge \neg(P \wedge R)\} \vee \neg\{P \wedge (Q \vee R)\}]$$

(6) $(\neg((R \vee \neg Q) \wedge \neg S) \wedge (\neg(\neg P \wedge Q) \wedge S))$



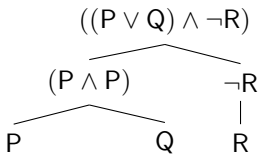
Main connective: $\wedge$

The wff can be made more readable, e.g. like this:

$$[\neg\{(R \vee \neg Q) \wedge \neg S\} \wedge \{\neg(\neg P \wedge Q) \wedge S\}]$$

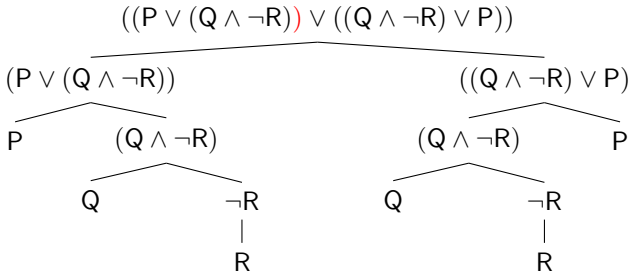
(b) *Which of the following expressions are wffs of a PL language with the relevant atoms? Repair the defective expressions by adding/removing the minimum number of brackets needed to do the job. Show the results are now wffs by producing parse trees.*

(1) $((P \vee Q) \wedge \neg R))$ is not balanced, i.e. the number of left and right brackets is not the same. Hence it can't be a wff. Delete the final bracket and we get the following wff:

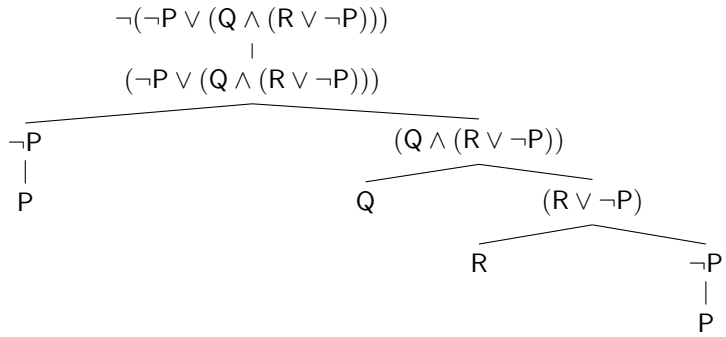


(2) $((P \vee (Q \wedge \neg R) \vee ((Q \wedge \neg R) \vee P))$ is not balanced.

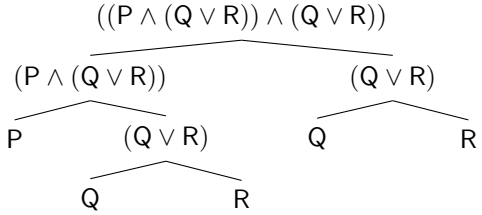
A simple repair (with the inserted bracket in red) is this:



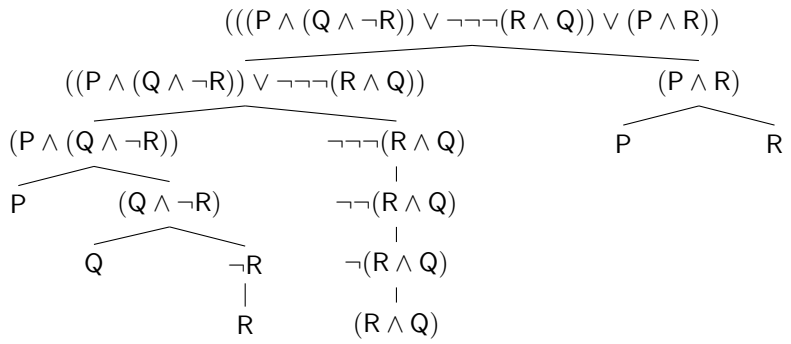
(3) $\neg(\neg P \vee (Q \wedge (R \vee \neg P))$ is once more unbalanced. We need to add a bracket at the end, to get



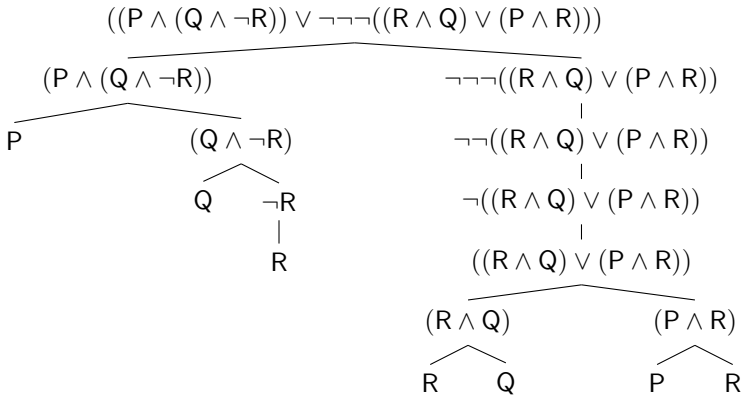
(4) $(P \wedge (Q \vee R) \wedge (Q \vee R))$ is balanced, but not a wff. It has the form of a three-way conjunction $(\alpha \wedge \beta \wedge \gamma)$. Some versions of PL languages allow this, but we have insisted on conjunctions always being two-way. So we need to bracket this as either $((\alpha \wedge \beta) \wedge \gamma)$ or $(\alpha \wedge (\beta \wedge \gamma))$. Going the first way, we get



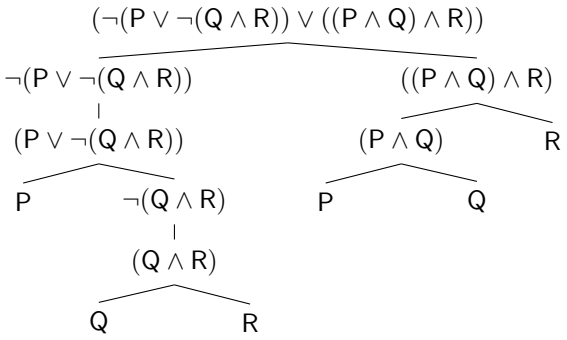
(5) $(((P \wedge (Q \wedge \neg R)) \vee \neg\neg\neg(R \wedge Q)) \vee (P \wedge R))$ is a wff as it stands.



(6) $((P \wedge (Q \wedge \neg R)) \vee \neg\neg\neg((R \wedge Q) \vee (P \wedge R)))$ is the same string of atoms and connectives as the last example, differently bracketed up. It is also a wff as it stands.



(7) $(\neg(P \vee \neg(Q \wedge R)) \vee (P \wedge Q \wedge R)$ is not balanced. The part $\alpha = \neg(P \vee \neg(Q \wedge R))$ is a wff; but the part $(P \wedge Q \wedge R)$ needs additional internal bracketing, one way or the other, to give us a wff β. The result though will be an expression of the form $(\alpha \vee \beta$, missing its final bracket. Add that and we get e.g.



(c*) *Show that parse trees for wffs are unique.*

(0) The *wffs* of a particular PL language are determined as follows. Having explicitly specified its atomic wffs, then:

(W1) Any atomic wff of the language counts as a wff.
(W2) If α and β are wffs, so is $(\alpha \wedge \beta)$.
(W3) If α and β are wffs, so is $(\alpha \vee \beta)$.
(W4) If α is a wff, so is $\neg\alpha$.
(W5) Nothing else is a wff.

The 'extremal' clause (W5) ensures that every wff must have *some* constructional history, some parse tree starting from atoms, recording a way it can be built up according to the principles (W2) to (W4).

One immediate consequence is that since brackets are always introduced in matching left/right pairs, every wff must have the same number of left-hand and right-hand brackets.

Suppose then that we look at a parse tree for a wff (at this point in the argument, we are not assuming uniqueness, just relying on the fact that there *is* at least one parse tree). When an occurrence to a binary connective, say $\wedge$, is first introduced at some point on a branch of this parse tree, it is in a (sub)formula of the form $(\alpha \wedge \beta)$, where α and β are wffs. And hence (since α is balanced), this connective $\wedge$ is preceded by one more left bracket than right bracket (and succeeded by one more right bracket than left bracket).

Now suppose that, as we go up the parse tree, this expression of the form $(\alpha \wedge \beta)$ becomes part of a longer formula formed using a binary connective, perhaps $((\alpha \wedge \beta) \vee \gamma)$ or $(\gamma \vee (\alpha \wedge \beta))$. In this sort of case, that occurrence of $\wedge$ will now be preceded by *two* more left brackets than right brackets (and succeeded by *two* more right brackets than left brackets). And as a binary connective gets buried deeper by the application of more connectives, it will acquire a greater excess of left brackets on its left (and symmetrically, a greater excess of right brackets on its right).

And so it goes. Generalizing, we have ...

(1) If a binary connective $\wedge$ or $\vee$ is the *main* connective of a wff of the form $(\alpha \wedge \beta)$ or $(\alpha \vee \beta)$ then the relevant occurrence of the connective '$\wedge$' or '$\vee$' is preceded by exactly one more left-hand bracket than right-hand bracket.

Any *other* occurrence of a binary connective in that wff will be preceded by at least two more left-hand brackets than right-hand brackets.

(2) You know that if a wff starts with a negation, it must have the form $\neg\alpha$, with α a wff.

And if it starts with a left bracket and ends with a right bracket, you now have a way of assigning it the form $(\alpha \wedge \beta)$ or $(\alpha \vee \beta)$, with α and β wffs – count brackets until you find the only binary connective which is preceded by exactly one more left bracket than right bracket.

(3) So now we have method of disassembling a complex wff stage by stage, building a parse tree downwards as you go. Here's one way of describing it:

(i) If a wff γ at a 'node' on the tree starts with a negation, it must have the form $\neg\alpha$; continue the branch of the tree downwards from that node by writing α beneath.

(ii) If a wff γ at a node on the tree starts with a left bracket and ends with a right bracket, it must have the form $(\alpha \wedge \beta)$ or $(\alpha \vee \beta)$. Then the relevant occurrence of $\wedge$ or $\vee$ is the only occurrence of a binary connective which is preceded by one more left bracket than right bracket. Find it! Take the preceding part of γ, minus its initial left bracket: that is to be α. Take the succeeding part of γ, minus its final right bracket: that is to be β. Then, from the node with γ, continue the parse tree by writing α beneath to the left, and β beneath to the right.

(iii) We can apply either rule (i) or rule (ii) to disassemble any wff. So applying on parsing complex wffs using these rules until you get down to the level of atoms.

There are no choice points in this process, hence if we indeed start with a wff, the result is the only possible one, and therefore parse trees are indeed unique.

Example: take the wff $(((\mathsf{P} \wedge (\mathsf{Q} \wedge \neg\mathsf{R})) \vee \neg\neg\neg(\mathsf{R} \wedge \mathsf{Q})) \vee (\mathsf{P} \wedge \mathsf{R}))$ again. Counting brackets, we find that the only binary connective preceded by exactly one more left bracket than right bracket is the second $\vee$. So our wff must be a disjunction, whose parse tree starts

$$\begin{array}{c} (((\mathsf{P} \wedge (\mathsf{Q} \wedge \neg\mathsf{R})) \vee \neg\neg\neg(\mathsf{R} \wedge \mathsf{Q})) \vee (\mathsf{P} \wedge \mathsf{R})) \\ \diagup \qquad\qquad \diagdown \\ ((\mathsf{P} \wedge (\mathsf{Q} \wedge \neg\mathsf{R})) \vee \neg\neg\neg(\mathsf{R} \wedge \mathsf{Q})) \qquad\qquad (\mathsf{P} \wedge \mathsf{R}) \end{array}$$

Again, counting brackets, the wff on the left branch is another disjunction, and so we know the parse tree must continue

$$\begin{array}{ccc} & (((\mathsf{P} \wedge (\mathsf{Q} \wedge \neg\mathsf{R})) \vee \neg\neg\neg(\mathsf{R} \wedge \mathsf{Q})) \vee (\mathsf{P} \wedge \mathsf{R})) & \\ & \diagup \qquad\qquad \diagdown & \\ ((\mathsf{P} \wedge (\mathsf{Q} \wedge \neg\mathsf{R})) \vee \neg\neg\neg(\mathsf{R} \wedge \mathsf{Q})) & & (\mathsf{P} \wedge \mathsf{R}) \\ \diagup \qquad\qquad \diagdown & & \\ (\mathsf{P} \wedge (\mathsf{Q} \wedge \neg\mathsf{R})) & \qquad \neg\neg\neg(\mathsf{R} \wedge \mathsf{Q}) & \end{array}$$

And so on we go.

(4) The same method (i)–(ii)–(iii) applied to any string of symbols can be used to decide whether it is a wff or not.

(i) If an expression γ at a 'node' is of the form $\neg\alpha$ (with α now not necessarily a wff), continue the branch of the tree downwards from that node by writing α beneath.

(ii) If an expression γ at a node on the tree stars with a left bracket and ending with a right bracket, then if possible find the unique occurrence of a binary connective which is preceded by one more left bracket than left bracket. If you can do this, then take the preceding part of γ, minus its initial left bracket: that is to be α (with α now not necessarily a wff). Take the succeeding part of γ, minus its final right bracket: that is to be β (with β now not necessarily a wff). Then, from the node with γ, continue the parse tree by writing α beneath to the left, and β beneath to the right.

(iii) Keep on parsing complex expressions until you can't apply (i) or (ii) any further.

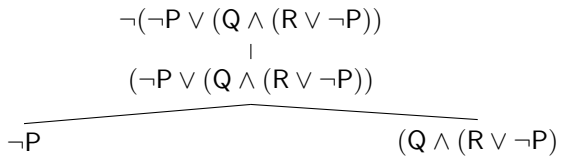
Either the process will generate a parse tree (and it must do if the expression is a wff). Or at some point the process fails, showing that the expression isn't a wff. There is no choice point in the process – it mechanically delivers a verdict about whether the original expression is a wff or not.

For example, take again the expression $((\mathsf{P} \vee (\mathsf{Q} \wedge \neg\mathsf{R}) \vee ((\mathsf{Q} \wedge \neg\mathsf{R}) \vee \mathsf{P}))$. Bracket-counting shows that no binary wff is preceded by exactly one more left than right bracket, so our parse-tree-building procedure fails the very first time we try to apply (ii).

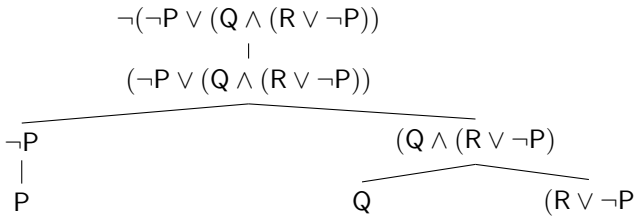
For another example, take again the expression $\neg(\neg\mathsf{P} \vee (\mathsf{Q} \wedge (\mathsf{R} \vee \neg\mathsf{P}))$. We can start building a parse tree like this:

$$\begin{array}{c} \neg(\neg\mathsf{P} \vee (\mathsf{Q} \wedge (\mathsf{R} \vee \neg\mathsf{P})) \\ | \\ (\neg\mathsf{P} \vee (\mathsf{Q} \wedge (\mathsf{R} \vee \neg\mathsf{P})) \end{array}$$

And bracket counting tells us to split the wff like this:



Continuing another step down each branch we get



But now the process freezes – neither rule (i) or (ii) applies to the rightmost branch. So the original expression isn't a wff.