1 Definition of input

Some symbols are defined in propositional logic. We call it language, all complicated strings are constructed based on these symbols with some rules.

**Definition 1.** The language of propositional logic consists of the following symbols:

1. **Connectives:** \( \lor, \land, \neg, \rightarrow, \leftrightarrow \)
2. **Parentheses:** ), ()
3. **Propositional Letters:** \( A, A_1, A_2, \cdots, B, B_1, B_2, \cdots \)

where we suppose the set of proposition letters is countable.

An inductive approach is defined to construct a proposition.

**Definition 2 (Proposition).** A proposition is a sequence of many symbols which can be constructed in the following approach:

1. **Propositional letters are propositions.**
2. If \( \alpha \) and \( \beta \) are propositions, then \( (\alpha \lor \beta), (\alpha \land \beta), (\neg \alpha), (\alpha \rightarrow \beta) \) and \( (\alpha \leftrightarrow \beta) \) are propositions.
3. A string of symbols is a proposition if and only if it can be obtained by starting with propositional letters (1) and repeatedly applying (2).

It is obvious that infinite propositions can be generated even if there are only finite proposition letters.

A proposition like \( ((A_1 \land A_2) \rightarrow B) \) is hard to represent with ascii character because of subscripts and connectives. Here we define some notations for propositions.

**Definition 3.** Every connective begins with a special character \( \backslash \), as following And proposition letter \( A_{ij} \) is represented as \( A_{\{ij\}} \).

Furthermore, the alphabet of proposition consist of only capitals. The subscript is just a natural number. In Table 1, subscript \( ij \) can be any letters. But We just let them be a concrete number for simplicity, e.g. natural number.

According to Definition 3, the proposition \( ((A_1 \land A_2) \rightarrow B) \) is a symbol sequence as \( ((A_{\{1\}} \land A_{\{2\}}) \land \rightarrow B) \).
Table 1: Notation of propositions

<table>
<thead>
<tr>
<th>connective</th>
<th>notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>\land</td>
<td>\land</td>
</tr>
<tr>
<td>\lor</td>
<td>\lor</td>
</tr>
<tr>
<td>\neg</td>
<td>\neg</td>
</tr>
<tr>
<td>\implies</td>
<td>\implies</td>
</tr>
<tr>
<td>\iff</td>
<td>\iff</td>
</tr>
</tbody>
</table>

2 Parsing algorithm

As a well-defined proposition can be uniquely mapped into a formation tree, which is easy to check every node well-defined or not. We can now introduce a recursive algorithm to analyze a sequence of symbols.

1. If all leaf nodes are labeled with proposition letters, stop it. Otherwise select a leaf node having expressions other than letter and examine it.
2. The first symbol must be (. if the second symbol is \neg, jump to step 4. Otherwise go to step 3.
3. (a) Scan the expression from the left until first reaching (\alpha, where \alpha is a nonempty expression having a balance between ( and ).
   (b) The \alpha is the first of the two constituents.
   (c) The next symbol must be \land, \lor, \implies, or \iff.
   (d) The remainder of the expression, \beta) must consist of a an expression \beta and ). Then we extend the tree by adding \beta as its immediate successor.
4. The first two symbols are now known to be (\neg. The remainder of the expression, \beta) must consist of a an expression \beta and ). Then we extend the tree by adding \beta as its immediate successor. Goto step 1.

In algorithm, we omit the procedure to exit for sequence which is not well-defined. Check every node, if a internal node is not well defined or terminal node is not a proposition letter, the algorithm just stops and asserts a non-well-defined sequence.

Given a sequence of symbols in Definition 3, the parsing algorithm should be deliberately modified to accommodate those representations. Furthermore, the depth of an input proposition is unknown in advance, the buffer should be allocated dynamically.

And you should be aware that the formation tree of a proposition is 2-branch and the length of proposition is not fixed in a node.
3 Visualization

The parsing algorithm can decide whether any sequence of symbols is well-defined or not. But it is hard for human to read it especially when a proposition is complicate enough.

Visualization can be introduced to help us understand easy with much more information. Here visualization can also help us to determine whether the implementation is correct. In another word, it also help us to debug your implementation of parsing algorithm.

A formation tree of a proposition \(((A\_1 \land A\_2) \implies B)\) is shown in Figure 1. There are two version, Figure 1b and Figure 1a. Each is more readable than \(((A\_1 \land A\_2) \implies B)\). In addition, Figure 1b is better than Figure 1a, which is more familiar to us.

The parsing algorithm can be modified to draw the formation tree of any given proposition during the parsing procedure. We suggest you to draw/output a node when an \(\lor\) or \(\land\) is determined.

4 Development environment

You can finish the project in language you wished. But we recommend Java and C. The Linux distribution Ubuntu 15.04 is a good choice, which is easy to install and use. It also provides GCC, Java, and related environment. For simplicity, you can install Ubuntu in a virtual machine. VirtualBox is free to obtain and works on Windows, Linux, and Mac OS.

5 Submission and grading

Finally, you should submit a package of you source code and a document describing you design. You should submit all belonging to school’s ftp site.

Whatever language is used, the document should describes how to build and deploy your program. If TA cannot run your program with your instruction, you will fail this project.

You running program take as input a file, which contains lines of propositions. You program just output yes or no and picture of tree if optional task is taken. You should construct a set of propositions to verify your implementation.

There are two subtask in this project. Parsing algorithm is mandatory. It is marked as 100.
Visualization is optional, which will make you have extra 40 as the bonus. A benchmark is used to grade your implementation. It contains positive and negative propositions. We will try to cover all cases. Some complicate propositions are also generated to test an input with arbitrary length.

Finally, you can contact us immediately if you have some question on the project.