Exercice 1

Rappelons quelques règles de la déduction naturelle, où A et B sont des formules logiques et Γ un ensemble de formules logiques quelconque :

$$\frac{}{\Gamma, A \vdash A}\text{ ax} \quad \frac{\Gamma \vdash \bot}{\Gamma \vdash A}\bot_e \quad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\to_i \quad \frac{\Gamma \vdash A \qquad \Gamma \vdash A \to B}{\Gamma \vdash B}\to_e \quad \frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\neg_i \quad \frac{\Gamma \vdash A \qquad \Gamma \vdash \neg A}{\Gamma \vdash \bot}\neg_e$$

1. Montrer que le séquent $\vdash \neg A \to (A \to \bot)$ est dérivable, en explicitant un arbre de preuve.
2. Montrer que le séquent $\vdash (A \to \bot) \to \neg A$ est dérivable, en explicitant un arbre de preuve.
3. Donner une règle correspondant à l'introduction du symbole $\wedge$ ainsi que deux règles correspondant à l'élimination du symbole $\wedge$. Montrer que le séquent $\vdash (\neg A \to (A \to \bot)) \wedge ((A \to \bot) \to \neg A)$ est dérivable.
4. On considère la formule $P = ((A \to B) \to A) \to A$ appelée *loi de Peirce*. Montrer que $\vDash P$, c'est-à-dire que P est une tautologie.
5. Pour montrer que le séquent $\vdash P$ est dérivable, il est nécessaire d'utiliser la règle d'absurdité classique $\bot_c$ (ou une règle équivalente), ce que l'on fait ci-dessous (il n'y aura pas besoin de réutiliser cette règle). Terminer la preuve du séquent $\vdash P$, dans laquelle on pose $\Gamma = \{(A \to B) \to A, \neg A\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{?}{\Gamma \vdash A}\,? \qquad \dfrac{}{\Gamma \vdash \neg A}\text{ ax}}{\Gamma \vdash \bot}\neg_e}{(A \to B) \to A \vdash A}\bot_c}{\vdash ((A \to B) \to A) \to A}\to_i$$

Exercice 2

La compilation du code compagnon initial avec `make safe` *provoque des avertissements attendus qui seront résolus lors de l'implémentation des fonctions demandées par le sujet.*

On s'intéresse dans cet exercice aux mots de Dyck, c'est-à-dire aux mots bien parenthésés. Dans ce type de mots, toute parenthèse ouverte « (» est fermée «) » et une parenthèse ne peut être fermée si elle ne correspond pas à une parenthèse préalablement ouverte.

Par exemple, pour deux couples de parenthèses, « (()) » et « ()() » sont des chaînes de parenthèses bien formées. « ())(» et «)()(» ne le sont pas.

On admet que le nombre de mots bien parenthésés à n couples de parenthèses est donné par les nombres de Catalan définis par la formule suivante :

$$C_n = \frac{(2n)!}{(n+1)!n!} \quad \text{pour } n \geqslant 0$$

On rappelle que le type `uint64_t` est un type entier non signé codé sur 64 bits.

1. Complétez dans le code compagnon la fonction dont le prototype est `uint64_t catalan(int n)`. Vous pouvez utiliser une fonction auxiliaire si cela vous semble pertinent.
2. Que va-t-il se passer si on tente d'afficher `catalan(n)` pour `n` un peu grand ? Le constatez-vous ici ?

On cherche maintenant à afficher le nombre de mots (chaînes) bien parenthésés avec n fixé couples de parenthèses, ainsi que les mots eux-mêmes.

Un algorithme de force brute pour déterminer toutes les chaînes à n couples de parenthèses bien formées consiste à générer toutes les possibilités puis à ne garder que les chaînes bien formées.

3. Complétez dans le code compagnon la fonction `bool verification(char * mot)`. Cette fonction renvoie `true` si le mot fourni en paramètre `mot` est bien parenthésé et `false` sinon.
4. Quelle est la complexité de cette vérification ?
5. Quelle est la complexité finale de l'algorithme de force brute ?

On appelle `n` le nombre de couples de parenthèses voulu. Dans le fichier compagnon fourni, le nombre de couples a été limité à 18.

On vous propose de coder l'énumération des chaînes de parenthèses bien formées en appliquant l'algorithme de backtracking suivant, dont on admet qu'il est correct : on compte le nombre de parenthèses ouvertes `o` et le nombre de parenthèses fermées `f` dans une chaîne de caractères courante (vide au départ).

- si `o = f = n`, on a trouvé une chaîne bien formée ;
- si `o < n`, on ajoute une parenthèse ouvrante et on relance ;
- si `f < o`, on ajoute une parenthèse fermante et on relance.

Cet algorithme est à implémenter dans la fonction `void dyck(char s[N], int o, int f, int n)` qui affiche sur la sortie standard les chaînes de parenthèses bien formées avec `n` couples de parenthèses lorsque `s` est la chaîne de caractère courante, `o` est son nombre de parenthèses ouvrantes et `f` son nombre de parenthèses fermantes.

6. Compléter la fonction `dyck` pour afficher les chaînes bien parenthésées avec 5 couples de parenthèses.
7. Adapter la fonction `dyck` pour calculer le nombre de mots obtenus. Combien de mots trouvez-vous pour 16 couples de parenthèses ?
8. Adapter la fonction `dyck` pour stocker les mots bien parenthésés dans une liste chaînée et les afficher après l'appel à la fonction. Vous trouverez dans le code compagnon une structure qui peut vous aider.