Logique et Déduction – Licence 2 – Termes et réécriture

UFR-ST – Université de Franche-Comté

Tous les exercices ne seront pas nécessairement traités en TD. Les chercher chez soi est une bonne façon d'approfondir les notions abordées ici sont à la base de nombreux concepts fondateurs en informatique. Vous pouvez questionner en fin (ou pendant si la situation le permet) le/la chargé(e) de TD sur les exercices non traités en cas de besoin.

Exercice 1. *(arbres et expressions)*
Dessiner les arbres correspondants aux formules suivantes :

1. $(A \vee B)$
2. $((A \vee B) \wedge (B \wedge C))$
3. $((\neg(A \vee B) \wedge (B \wedge \neg C)) \Rightarrow (A \wedge \neg B))$

Exercice 2. *(écriture notation Polonaise inverse)*
Encire en notation Polonaise inverse (postfixée) les expressions suivantes :

1. $(A \vee B)$
2. $((A \vee B) \wedge (B \wedge C))$
3. $((\neg(A \vee B) \wedge (B \wedge \neg C)) \Rightarrow (A \wedge \neg B))$

Exercice 3. *(évaluation d'expression en Polonaise inverse)*
Évaluer chacune des expressions arithmétiques suivantes écrite en Polonaise inverse :

1. 3 4 +
2. 5 3 4 + −
3. 6 5 3 + 5 − ∗ 3 2 + −
4. 1 2 − 3 4 ∗ 5 − 6 7 + + +

Exercice 4. *(transformation en Polonaise inverse)*
Écrire une fonction Python qui prend une liste contenant des entiers et des symboles dans `+`, `-` et `*` codant une expression arithmétique sous forme polonaise inverse et qui l'évalue (comme dans l'exercice précédent). On ne vérifiera pas que l'expression est correcte. On rappelle qu'on peut en Python supprimer l'élément d'indice `i` d'une liste `L` par l'instruction

```
del L[i]
```

Exercice 5. *(parcours préfixe d'un terme/arbre)*
On va étudier dans cet exercice le *parcours préfixe d'un terme* (ou d'un arbre) qui est une méthode générale pour visiter tous les éléments d'un arbre (par exemple pour les compter, les énumérer, etc.). C'est la façon dont fonctionne la commande `ls -R` sous linux par exemple. Le principe algorithmique est le suivant :

```
Parcours-Prefixe(terme):
   Traiter(racine(terme))
   Pour chaque sout terme t de terme:
      Parcours-Prefixe(t)
   FinPour
```

Si l'on veut, par exemple, afficher tous les symboles apparaissant dans un terme, le traitement sera un `print` de chaque élément visité et on aura :

```
Parcours-Prefixe-Afficher(terme):
   print(symbole(racine(terme)))
   Pour chaque sout terme t de terme:
      Parcours-Prefixe-Afficher(t)
   FinPour
```

Si l'on veut compter le nombre de constantes `A` dans un terme on aura :

```
Parcours-Prefixe-CompterA(terme):
   Si terme == A:
      return 1
   res = 0
   Pour chaque sout terme t de terme:
      res+=Parcours-Prefixe-CompterA(t)
   FinPour
   return res
```

1. Que donne l'exécution de `Parcours-Prefixe-Afficher` sur le terme $f(A, B)$.
2. Que donne l'exécution de `Parcours-Prefixe-Afficher` sur le terme $h(f(A, f(B, A)))$.
3. Que donne l'exécution de `Parcours-Prefixe-Afficher` sur le terme
$$h(f(h(f(h(A), f(B, A))), g(A, B))).$$
4. Écrire, en pseudo-code, un algorithme utilisant un parcours préfixe pour calculer le nombre de symboles dans un terme.
5. Écrire, en pseudo-code, un algorithme utilisant un parcours préfixe pour calculer le nombre de symbole f (qui n'est pas une constante) dans un terme.

Exercice 6. *(parcours suffixe d'un terme/arbre)*
On va étudier dans cet exercice le *parcours suffixe d'un terme* (ou d'un arbre) qui est une méthode générale pour visiter tous les éléments d'un arbre (par exemple pour les compter, les énumérer, etc.). Le principe algorithmique est le suivant :

```
Parcours-Suffixe(terme):
   Pour chaque sout terme t de terme:
      Parcours-Suffixe(t)
   FinPour
   Traiter(racine(terme))
```

Si l'on veut, par exemple, afficher tous les symboles apparaissant dans un terme, le traitement sera un `print` de chaque élément visité et on aura :

```
Parcours-Suffixe-Afficher(terme):
   Pour chaque sout terme t de terme:
      Parcours-Suffixe-Afficher(t)
   FinPour
   print(symbole(racine(terme)))
```

1. Que donne l'exécution de `Parcours-Suffixe-Afficher` sur le terme $f(A, B)$.
2. Que donne l'exécution de `Parcours-Suffixe-Afficher` sur le terme $h(f(A, f(B, A)))$.

3. Que donne l'exécution de `Parcours-Suffixe-Afficher` sur le terme

$$h(f(h(f(h(A), f(B, A))), g(A, B))).$$

Exercice 7. *(positions d'un terme)*
Donner, pour les termes t suivants les ensembles $\text{var}(t)$ et $\text{pos}(t)$. L'alphabet est $\{A, B, C, h, f, g\}$ où A, B, C sont des constantes, h est d'arité 1 et f et g sont d'arité 2.

1. $t = A$
2. $t = x$
3. $t = h(A)$
4. $t = h(x)$
5. $t = h(h(h(f(x, h(y)))))$
6. $t = h(f(h(f(h(y), f(B, x))), g(x, B)))$.

Exercice 8. *(termes et positions)*
L'alphabet est $\{A, B, C, h, f, g\}$ où A, B, C sont des constantes, h est d'arité 1 et f et g sont d'arité 2. Donner dans chacun des cas ci-dessous un terme qui a exactement l'ensemble donné pour ensemble de positions (plusieurs solutions sont possibles).

1. $\{\varepsilon, 1, 1 \cdot 1, 1 \cdot 2\}$
2. $\{\varepsilon, 1, 1 \cdot 1, 1 \cdot 1 \cdot 1, 1 \cdot 1 \cdot 2\}$
3. $\{\varepsilon, 1, 1 \cdot 1, 1 \cdot 2, 1 \cdot 1 \cdot 1, 1 \cdot 1 \cdot 2, 1 \cdot 2 \cdot 1, 1 \cdot 2 \cdot 1 \cdot 1, 1 \cdot 2 \cdot 1 \cdot 1 \cdot 1\}$.

Exercice 9. *(sous-termes)*
On considère le terme :

$$t = h(f(h(f(h(y), f(B, x))), g(x, B))).$$

1. Que vaut $t_{|\varepsilon}$?
2. Que vaut $t_{|1}$?
3. Que vaut $t_{|1 \cdot 2}$?
4. Que vaut $t_{|1 \cdot 1 \cdot 1}$?
5. Que vaut $t_{|1 \cdot 1 \cdot 1 \cdot 2}$?

Exercice 10. *(codage machine des formules)*

1. En utilisant le codage des formules de logique propositionnelle vu en Cours et TP, donner un code qui construit la formule
   `AND(X0,NOT(X1))`
   sans utiliser la fonction `fromString`.
2. Dessiner la structuration mémoire de cette formule.

Exercice 11. *(codage machine et nombre de négations)*
En utilisant le codage des formules de logique propositionnelle vu en Cours et TP, donner un algorithme qui retourne le nombre de négations utilisées dans une formule.

Exercice 12. *(codage machine et positions)*
En utilisant le codage des formules de logique propositionnelle vu en Cours et TP, donner un algorithme qui retourne l'ensemble des positions du terme. Cet ensemble sera codé dans une liste. Chaque position sera une chaîne de caractères utilisant des `1` et des `2`.

Exercice 13. *(codage machine et écriture en Polonaise inverse)*
En utilisant le codage des formules de logique propositionnelle vu en Cours et TP, donner un algorithme qui retourne une chaîne de caractère de la formule en notation Polonaise inverse. On pourra utiliser un parcours suffixe.

Exercice 14. *(substitution dans un terme)*

On considère le terme :

$$t = h(f(h(f(h(y), f(B, x))), g(x, B))).$$

1. Que vaut $t[f(A, B)]_{|\varepsilon}$?
2. Que vaut $t[h(B)]_1$?
3. Que vaut $t[C]_{1\cdot 2}$?
4. Que vaut $t[t_{1\cdot 1}]_{1\cdot 1\cdot 1}$?
5. Que vaut $t[h(A)]_{1\cdot 1}[f(A, B)]_{1\cdot 1\cdot 1}$?

Exercice 15. *(substitution dans un terme variable)*

On considère le terme :

$$t = h(f(h(f(h(y), f(B, x))), g(x, B))).$$

1. En supposant que $\sigma(x) = h(A)$ et $\sigma(y) = f(A, B)$, que vaut $t\sigma$?
2. En supposant que $\sigma(x) = f(h(A), h(B))$ et $\sigma(y) = h(f(A, B))$, que vaut $t\sigma$?

Exercice 16. *(réécriture de termes)*

L'alphabet est $\{A, B, C, h, f, g\}$ où A, B, C sont des constantes, h est d'arité 1 et f et g sont d'arité 2. On considère le terme

$$t = f(h(A), h(h(h(A)))).$$

1. Donner un terme t_1 tel que $t \to_{h(h(x))\to h(x)} t_1$. On précisera la substituion de variables utilisée ainsi que la position de la réécriture.
2. Donner explicitement l'ensemble $\{t' \mid t \to^*_{h(h(x))\to h(x)} t'\}$.
3. On considère le système de réécriture $\mathcal{R} = \{h(h(x)) \to h(x), f(h(x), h(x)) \to g(h(x), h(h(x)))\}$. Donner explicitement $\{t' \mid t \to^*_{\mathcal{R}} t'\}$.
4. Redonner la définition d'un système de réécriture *terminant*. Le système $\mathcal{R}$ est-il terminant ? Justifier (on ne demande pas une preuve mathématique rigoureuse, juste des arguments).

Exercice 17. *(formes normales)*

On considère le système de réécriture vu en cours

$$\mathcal{R} = \{\neg(\neg(x)) \to x, \neg(\wedge(x, y)) \to \vee(\neg(x), \neg(y)), \neg(\vee(x, y)) \to \wedge(\neg(x), \neg(y))\}.$$

Donner la forme normale de chacune des formules suivantes :

1. $\neg(\wedge(A, B))$.
2. $\neg\neg(\wedge(A, B))$, en commençant par la règle $\neg(\neg(x)) \to x$.
3. $\neg\neg(\wedge(A, B))$, en commençant par la règle $\neg(\wedge(x, y)) \to \vee(\neg(x), \neg(y))$.
4. $\neg(\wedge(\neg(A), \neg(\vee(\vee(A, \neg(B)), \neg(B)))))$.

Exercice 18. *(simplification de formules)*

On considère le système de réécriture $\mathcal{R}$ constitué des règles suivantes :

- $\wedge(\top, x) \to x$
- $\wedge(\bot, x) \to \bot$

- $\vee(\top, x) \to \top$
- $\vee(\bot, x) \to x$
- $\vee(x, x) \to x$
- $\wedge(x, x) \to x$

1. Pourquoi peut-on appeler ces règles des *règles de simplification* ?
2. En les utilisant simplifier la formule

$$\vee(\wedge(\vee(B, A), \vee(B, A)), \wedge(A, \top))$$

3. Proposer d'autres règles de simplification.

Remarque : Pour des algorithmes sur des formules logiques coûteux en temps ou en espace, il est souvent très efficace de commencer par simplifier les formules pour augmenter l'efficacité. Attention cependant à ne pas introduire des règles (par exemple pour gérer la commutativité ou l'associativité) qui ne simplifient pas et peuvent faire exploser le temps de pré-traitement.

Exercice 19. *(entiers de Peano)*

Les entiers naturel de Peano sont définis par des termes sur l'alphabet contenant deux symboles, 0 (constante) et S qui est unaire. On empile simplement un nombre S correspondant à l'entier que l'on souhaite coder ; il s'agit d'une écriture en unaire. Par exemple, 3 s'écrira avec le terme

$$S(S(S(0))).$$

L'addition est codé par l'introduction d'un symbole binaire A (pour addition) et par les deux règles de réécriture d'un système $\mathcal{R}$:

- $A(x, 0) \to x$,
- $A(x, S(y)) \to A(S(x), y)$.

1. Donner une forme normale (pour $\mathcal{R}$) de $A(S(S(0)), 0)$.
2. Donner une forme normale (pour $\mathcal{R}$) de $A(0, S(S(0)))$.
3. Donner une forme normale (pour $\mathcal{R}$) de $A(S(S(S(0))), S(S(0)))$.
4. Donner une forme normale (pour $\mathcal{R}$) de $A(S(S(S(0))), A(S(S(0)), S(S(0))))$.
5. Justifier brièvement que $\mathcal{R}$ est terminant.

Exercice 20. *(réécriture et calcul formel)*
La réécriture est beaucoup utilisé en calcul formel. On considère (pour simplifier) l'alphabet

$$\Sigma = \{0, 1, \ldots, 9, -, \cos, \sin, \exp, *, +, \mathrm{var}\},$$

où $0, \ldots, 9$ sont des constantes, $-, \cos, \sin, \exp$ sont unaires, et $*$ et $+$ sont binaires.

La fonction à une variable $(x + \sin x)e^{\cos x}$ sera codée (on ne tient pas compte des codages des ensembles de définition) par le terme $*(+(\mathrm{var}(x), \sin(\mathrm{var}(x))), \exp(\cos(\mathrm{var}(x))))$.

On introduit un nouveau symbole unaire d pour modéliser la dérivation. Par exemple, $((x + \sin x)e^{\cos x})'$ sera codé par $d(*(+(x, \sin(x)), \exp(\cos(x))))$.

1. Donner des règles de réécriture codant le fait que la dérivée d'une constante de $\{0, \ldots, 9\}$ est nulle.
2. Donner une règle de réécriture codant le fait que la dérivée d'une variable et 1.
3. Donner une règle de réécriture pour la dérivée d'une somme.
4. Donner une règle de réécriture pour la dérivée d'un produit.
5. Donner une règle de réécriture pour la dérivée de cos, de sin, exp.

6. En dessinant l'arbre du terme de $d(*(+(\text{var}(x), \sin(\text{var}(x))), \exp(\cos(\text{var}(x)))))$ et en utilisant les règles définies, calculer cette dérivée sous forme d'arbre.

Remarque : Les logiciels de calcul formel utilisent ensuite des règles (et de stratégie) de réécriture pour simplifier les expression obtenues.

Exercice 21. *(L-système (générateur d'arbres))*

Le paradigme de L-système est un paradigme de réécriture (formalisé un peu différemment) permettant des modélisations par exemple de croissance d'arbre. Nous allons ici en donner une traduction en réécriture (comme vu en cours).

On considère l'alphabet $\{A, F, t, b\}$ où A et F sont des constantes, t est unaire, et b est binaire. On considère aussi le système de réécriture

$$\mathcal{R} = \{A \to t(A), A \to F, A \to B(A, A)\}.$$

1. Donner cinq terme t différents et irréductibles tels que $A \to_{\mathcal{R}}^* t$.
2. $\mathcal{R}$ est-il terminant ? confluent ?
3. En quoi cela peut-il être utilisé pour être un générateur d'arbres (biologique) ?

Remarque : Au delà de la modélisation biologique, ce type de système est aussi utilisé pour la génération d'arbres dans les jeux vidéo. Pour cela, on ajoute des probabilités et de systèmes bien plus étoffés.

Exercice 22. *(réécriture sur les mots)*
La réécriture, vue en cours, est sur des termes/arbres. Il est aussi possible de la faire sur des structures plus complexes (graphes) mais aussi sur des structures plus simples comme les mots (ou chaînes de caractère).

Sans redéfinir tout formellement, un système de réécriture de mots (appelé aussi Semi-Thue-System) sur un alphabet (ensemble de caractères) est un ensemble fini de règle du type

$$u \to v,$$

où u et v sont des mots.

Réécrire un mot w par la règle $u \to v$ consiste à rechercher une occurrence de u dans w et à la remplacer par v,à la manière d'un *query-replace* appliqué une seule fois.

Considérons par exemple $\Sigma_0 = \{a, b, c\}$, et le système $\mathcal{R}_0 = \{aa \to a, bb \to b, cc \to c\}$. Le mots *abacb***aa***ba***cc** peut se récrire en *abacbabacc* ou *abacbaabac*.

Les notions de *confluence*, *terminaison*, *irréductibilité* ou *forme normale* se définissent comme sur les termes.

La réécriture de mot est particulièrement utilisée dans la manipulation des textes.

1. Justifier que $\mathcal{R}_0$ est terminant et confluent.
2. Comment peut-on définir par une phrase en français l'unique forme normal d'un mot w pour $\mathcal{R}_0$?
3. On considère l'alphabet $\Sigma_1 = \{a, \overline{a}, b, \overline{b}, \varepsilon\}$ et le système $\mathcal{R}_1 = \{a\overline{a} \to \varepsilon, \overline{a}a \to \varepsilon, b\overline{b} \to \varepsilon, \overline{b}b \to \varepsilon, a\varepsilon \to a, \varepsilon a \to a, b\varepsilon \to b, \varepsilon b \to b\}$. Donner une forme normale de $a\overline{a}a\overline{b}a\overline{a}bb\overline{b}abab$.
4. Justifier que $\mathcal{R}_1$ est terminant. On peut aussi démontrer (ne pas le faire) qu'il est confluent. Ce système est utilisé en algèbre, il définit le groupe libre engendré par deux éléments.

Exercice 23. *(réécriture de mots, distance de Levenshtein)*
Pour traiter cet exercice il faut utiliser les notions définies dans l'exercice 22.

Connaître la ressemblance de deux chaînes de caractères est utilisée en génétique pour comparer des gènes (qui subissent des mutations) ou pour les correcteurs orthographiques (type envoi de SMS) pour trouver le ou les mots proches du dictionnaires.

Soit $\Sigma = \{a_1, \dots, a_n\}$ un alphabet (ensemble de caractères). On définit le système de réécriture $\mathcal{R}$ suivant sur les mots sur Σ :

- $a_i \to a_j$, pour tout $i \neq j$, qui code le remplacement d'une lettre par une autre.
- $a_i \to \varepsilon$, pour tout i ; ε désignant la chaîne de caractères vide. Cette règle code l'effacement (ou l'oublie) d'une lettre.
- $\varepsilon \to a_i$ pour tout i, qui code l'insertion d'une lettre.

Par exemple, on a la suite de réécritures suivante qui est possible :

$$\text{maison} \to \text{masson} \to \text{mosson} \to \text{moisson} \to \text{poisson} \to \text{poison}$$

1. Justifier que pour tout couple de mots u, v sur Σ, on a $u \to^* v$.
2. Trouver une façon de réécrire *maison* en *poison* en deux étapes seulement.
3. La distance de Levenshtein[1], dit aussi distance d'édition, est définit sur les mots sur Σ comme étant le nombre minimal de réécriture qu'il faut faire pour réécrire un mot en un autre. La distance de Levenshtein entre u et v sera noté $d(u, v)$. Que vaut $d(\text{maison}, \text{poison})$?
4. Justifier que si $d(u, v) = 0$ alors $u = v$.
5. Justifier que $d(u, v) = d(v, u)$, pour tous mots u, v.
6. Justifier que $d(u, v) \leq d(u, w) + d(w, v)$ pour tous mots v, v, w.
7. Justifier que $d(u, v) \leq |u| = |v|$ où $|\cdot|$ désigne la longueur du mot (nombre de lettres).
8. Proposer une idée d'algorithme pour calculer $d(u, v)$. On ne s'intéressera pas à l'efficacité ici, mais on prendra soin à la terminaison (pas de boucle infini).
9. On pose $u = u_1 \dots u_k$ et $v = v_1 \dots v_n$ où les u_i et v_j sont des lettres. On pose

$$\text{lev}_{u,v}(i, j) = d(u_1 \dots u_i, v_1 \dots v_j),$$

   c'est-à-dire la distance des i premières lettres de u (éventuellement aucune si $i = 0$) avec les j premières lettres de v (éventuellement $j = 0$). Que vaut $\text{lev}_{u,v}(1, 1)$ si u et v ont chacun au moins une lettre ?
10. Que vaut $\text{lev}_{u,v}(|u|, |v|)$?
11. Que vaut $\text{lev}_{u,v}(0, j)$? $\text{lev}_{u,v}(i, 0)$?
12. Justifier que $d(ua, v) \leq d(u, v) + 1$. En déduire que $\text{lev}_{u,v}(i - 1, j) \leq lev(i, j) + 1$.
13. Justifier que $d(ua, va) \leq d(u, v)$ et que $d(ua, vb) \leq d(u, v) + 1$ si $a \neq b$.
14. On admettra les égalités suivantes :

$$\begin{cases} \text{lev}_{u,v}(i, 0) = i \\ \text{lev}_{u,v}(0, j) = j \\ \\ \text{Si } i \neq 0 \text{ et } j \neq 0, \text{ alors } \text{lev}_{u,v}(i, j) = \min \begin{cases} \text{lev}_{u,v}(i - 1, j) + 1 \\ \text{lev}_{u,v}(i, j - 1) + 1 \\ \text{lev}_{u,v}(i - 1, j - 1) + 1 - \delta_{u_i, u_j} \end{cases} \end{cases},$$

    où $\delta_{x,y} = 1$ si $x = y$ et 0 sinon. En utilisant ce résultat, calculer $d(\text{poire}, \text{rois})$.
15. En utilisant les égalités précédentes, proposer un algorithme efficace pour calculer $d(u, v)$.

1. Il existe aussi une distance de Damereau-Levenshtein qui prend en compte les permutations de lettres.