Réécriture et partage

Épreuve pratique d'algorithmique et de programmation
Concours commun des Écoles normales supérieures

Durée de l'épreuve : 3 heures 30 minutes

Juin/Juillet 2024

ATTENTION !

N'oubliez en aucun cas de recopier votre u_0
à l'emplacement prévu sur votre fiche réponse

Important.

Il vous a été donné un numéro u_0 qui servira d'entrée à vos programmes. Les réponses attendues sont généralement courtes et doivent être données sur la fiche réponse fournie à la fin du sujet. À la fin du sujet, vous trouverez en fait deux fiches réponses. La première est un exemple des réponses attendues pour un $\widetilde{u_0}$ particulier (précisé sur cette même fiche et que nous notons avec un tilde pour éviter toute confusion !). Cette fiche est destinée à vous aider à vérifier le résultat de vos programmes en les testant avec $\widetilde{u_0}$ au lieu de u_0. Vous indiquerez vos réponses (correspondant à votre u_0) sur la seconde et vous la remettrez à l'examinateur à la fin de l'épreuve.

En ce qui concerne la partie orale de l'examen, lorsque la description d'un algorithme est demandée, vous devez présenter son fonctionnement de façon schématique, courte et précise. Vous ne devez en aucun cas recopier le code de vos procédures !

Quand on demande la complexité en temps ou en mémoire d'un algorithme en fonction d'un paramètre n, on demande l'ordre de grandeur en fonction du paramètre, par exemple : $O(n^2)$, $O(n \log n)$,...

Il est recommandé de commencer par lancer vos programmes sur de petites valeurs des paramètres et de ***tester vos programmes sur des petits exemples que vous aurez résolus préalablement à la main ou bien à l'aide de la fiche réponse type fournie en annexe***. Enfin, il est recommandé de lire l'intégralité du sujet avant de commencer afin d'effectuer les bons choix de structures de données dès le début.

Les Parties 1 et 3 (OCaml) sont indépendantes de la Partie 2 (C).

Le fichier `base.ml` correspond aux Parties 1 et 3, et le fichier `base.c` à la Partie 2. Le dossier `data/` contient les jeux de tests.

Il est recommandé de garder une sauvegarde de tout les fichiers fournis, au cas où ceux-ci seraient modifiés par erreur.

Il est précisé dans chaque partie le langage d'implémentation à utiliser. Il est demandé de nous fournir sur votre clef USB vos fichiers OCaml et C. Ces consignes doivent impérativement être suivies.

Préliminaires

Jeux de tests et entrées Plusieurs jeux de tests sont disponibles dans le répertoire `data/`. Chacun se trouve dans un répertoire `data/U0/`, où `U0` est la valeur u_0. Ces jeux de tests sont utilisés dans les Parties 1 et 3. Des fonctions en OCaml `read_terms`, `read_rule_terms` et `read_rules` servant à la lecture depuis les fichiers sont fournies dans le fichier `base.ml`.

Résultats et évaluation Votre évaluation portera sur les fichiers dont l'`U0` correspond à votre u_0. Les autres fichiers peuvent être utilisés pour tester vos programmes.

Notations On rappelle que pour deux entiers naturels a et b, avec $b \neq 0$, $(a \mod b)$ désigne le reste de la division euclidienne de a par b, c'est-à-dire l'unique entier r avec $0 \leq r < b$ tel qu'il existe un $k \in \mathbb{N}$ vérifiant $a = k \times b + r$.

1 Partie 1 (OCaml) : Réécriture de termes

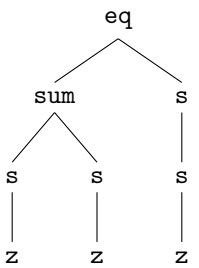
Vous devez utiliser OCaml tout au long de cette partie.

Code fourni Le fichier `base.ml` contient les éléments de code suivants :

- Un module **`Tag`** permettant de créer des tags de façon abstraites.
- Les déclarations des types `term`, `rule_term` et `rule`.
- Les fonctions `read_terms`, `read_rule_terms` et `read_rules` permettant de lire les fichiers de données dans `data/U0`.
- Un module **`HashCons`** utilisé dans la Partie 3.

Les modules, types et fonctions fournis sont décrits plus bas lorsque ceux-ci sont utilisés.

Termes Un terme est un arbre non-vide dont les nœuds sont étiquetés par des symboles (représentés par des chaînes de caractères). Un nœud interne est un nœud avec au moins un enfant, et une feuille un nœud sans enfant. Par exemple, le terme t_0 suivant :



a trois feuilles, toutes étiquetées par `z`, et six nœuds internes étiquetés par `s`, `sum` et `eq`.
Un terme peut aussi être représenté par un mot, dans lequel chaque nœud du terme est suivi par la liste de ses enfants, entre parenthèses et séparés par des virgules. Ainsi, une feuille étiquetée par `c` (n'ayant donc pas d'enfant) est représentée par `c()`. On s'autorisera à enlever les parenthèses pour les symboles sans enfant, et à écrire `c` à la place de `c()`. Par exemple, le terme t_0 peut être représenté par :

$$\texttt{eq}(\texttt{sum}(\texttt{s}(\texttt{z}), \texttt{s}(\texttt{z})), \texttt{s}(\texttt{s}(\texttt{z}))) \tag{1}$$

Chaque nœud d'un terme s définit un sous-terme de s : par exemple, `z` et $\texttt{sum}(\texttt{s}(\texttt{z}), \texttt{s}(\texttt{z}))$ sont des sous-termes de t_0. Notez qu'un terme s est un sous-terme de lui-même.

On utilisera le type OCaml suivant pour représenter les termes :

```
type term = Tfun of Tag.t * string * term list
```

où la construction OCaml `Tfun (tag, symb, [t1 ; ...; tn])` représente le terme `symb(t1, ..., tn)`, dont la racine est étiquetée par `symb` et possède n fils `t1`, ..., `tn`. Le tag `tag` ne sera pas utilisé avant la Section 3. Pour l'instant, on utilisera le tag par défaut `Tag.default` à chaque fois qu'un tag est nécessaire. En OCaml, le terme t_0 est représenté par :

```
let d = Tag.default in
let one = Tfun (d, "s", [Tfun (d, "z", [])]) in  (*   s(z)   *)
let two = Tfun (d, "s", [one])                in  (* s(s(z)) *)
Tfun (d, "eq", [Tfun (d, "sum", [one; one]); two])
```

Question 1 *Implémenter la fonction de signature* `val hash : term -> int`*, telle que pour tout terme* $\texttt{f}(\texttt{t}_1, \ldots, \texttt{t}_n)$ *:*

$$\texttt{hash}\big(\texttt{f}(\texttt{t}_1, \ldots, \texttt{t}_n)\big) = \Big(\big(\texttt{Hashtbl.hash f}\big) + \textstyle\sum_{i=1}^{n} i \times \texttt{hash}(\texttt{t}_i)\Big) \mod 104\,729$$

où la fonction `Hashtbl.hash` *est fournie par la bibliothèque standard OCaml.*

Notons `terms = read_terms "data/U0/q1.txt"` *(en remplaçant* `U0` *par votre* u_0*).*
Calculer `hash (List.nth terms k)` *pour les valeurs de* `k` *suivantes :*

a) $k = 0$ ***b)*** $k = 1$ ***c)*** $k = 2$ ***d)*** $k = 3$

1.1 Substitution et *matching*

Un terme de règle est un terme dont certaines feuilles sont étiquetées par des variables, prises dans l'ensemble de variables $\mathcal{V} = \{\texttt{x}_0, \texttt{x}_1, \ldots, \texttt{x}_9\}$. Dans ce sujet, nous n'aurons jamais besoin de plus de 10 variables. Par exemple, $\texttt{sum}(\texttt{x}_0, \texttt{s}(\texttt{x}_1))$ et $\texttt{eq}(\texttt{x}_0, \texttt{x}_0)$ sont deux termes de règle. En OCaml, on représentera une variable $\texttt{x}_i$ par l'entier i, et les termes de règle par le type :

```
type rule_term =
  | Rvar of int (** on représente chaque variable par un entier *)
  | Rfun of string * rule_term list
```

Ainsi, $\texttt{sum}(\texttt{x}_0, \texttt{s}(\texttt{x}_1))$ est représenté par `Rfun("sum", [Rvar 0; Rfun ("s", [Rvar 1])])`.
Une substitution est une fonction associant des termes à des variables. Par exemple,

$$\sigma_0 = \big\{\texttt{x}_0 \mapsto \texttt{s}(\texttt{z});\ \ \texttt{x}_1 \mapsto \texttt{z};\ \ \texttt{x}_2 \mapsto \texttt{h}(\texttt{b}, \texttt{s}(\texttt{z}))\big\} \tag{2}$$

associe $\texttt{s(z)}$ à $\texttt{x}_0$, $\texttt{z}$ à $\texttt{x}_1$, $\texttt{h(b, s(z))}$ à $\texttt{x}_2$, et rien aux autres variables.
Naturellement, on peut appliquer une substitution σ à un terme de règle s pour obtenir un terme $\sigma(s)$, en remplaçant chaque variable $\texttt{x}_i$ apparaissant dans s par $\sigma(\texttt{x}_i)$. Par exemple :

$$\sigma_0(\texttt{sum}(\texttt{x}_0, \texttt{s}(\texttt{x}_1))) = \texttt{sum}(\texttt{s}(\texttt{z}), \texttt{s}(\texttt{z}))$$

Si une variable $\texttt{x}_i$ apparaissant dans le terme de règle n'est associée à aucun terme par σ, on substituera celle-ci par le terme `missing()`.

Question 2 *Choisir un type de données, que l'on appellera* `subst`*, pour représenter les substitutions. Implémenter la fonction appliquant une substitution à un terme de règle, de signature* `val subst : subst -> rule_term -> term`.

Notons `rule_terms = read_rule_terms "data/U0/q2.txt"`.
Soit `sigma0` *la substitution* σ_0 *donnée dans l'équation* (2) *ci-dessus. Calculer* `hash (subst sigma0 (List.nth rule_terms k))` *pour les valeurs de* `k` *suivantes :*

a) $k = 0$ ***b)*** $k = 1$ ***c)*** $k = 2$ ***d)*** $k = 3$

Question à développer pendant l'oral 1 *Donner la complexité en temps de votre algorithme.*

Soit t un terme et l un terme de règle. On dit que t *matche* l lorsqu'il existe une substitution σ telle que $t = \sigma(l)$. Par exemple :

$\texttt{sum}(\texttt{s}(\texttt{z}), \texttt{s}(\texttt{z}))$	*matche*	$\texttt{sum}(\texttt{x}_0, \texttt{s}(\texttt{x}_1))$	avec $\sigma = \{\texttt{x}_0 \mapsto \texttt{s}(\texttt{z});\ \texttt{x}_1 \mapsto \texttt{z}\}$
$\texttt{eq}(t, t)$	*matche*	$\texttt{eq}(\texttt{x}_0, \texttt{x}_0)$	avec $\sigma = \{\texttt{x}_0 \mapsto t\}$
$\texttt{eq}(t, t')$	*ne matche pas*	$\texttt{eq}(\texttt{x}_0, \texttt{x}_0)$	quand $t \neq t'$

Question 3 *Implémenter une fonction de signature :*

```
val find_match : term -> rule_term -> subst option
```

telle que si `t` *matche* `l`*, alors* `find_match t l` *renvoie* `Some sigma` *pour un* `sigma` *tel que* `t = subst sigma l`*, et sinon* `find_match t l` *renvoie* `None`.

Notons `terms = read_terms "data/U0/q3.txt"`.
Soit `t` *le k-ième terme de* `terms`*. Si* `find_match t r = Some sigma`*, calculer la valeur* `v` *de* `hash (subst sigma (Rvar 0))) + hash (subst sigma (Rvar 1)))` *et noter* `Some v` *sur la fiche réponse (sinon, noter* `None`*) pour les valeurs de* k *et* `r` *suivantes :*

a) $k = 0$ *et* $r = \texttt{eq}(x_0, x_0)$ ***b)*** $k = 1$ *et* $r = \texttt{eq}(x_0, x_0)$

c) $k = 2$ *et* $r = \texttt{F}(\texttt{f}(x_0), x_1, \texttt{g}(x_1))$ ***d)*** $k = 3$ *et* $r = \texttt{F}(\texttt{f}(x_0), x_1, \texttt{g}(x_1))$

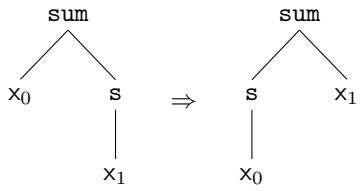
On dit qu'un terme de règle `l` est linéaire si aucune variable $\texttt{x}_i$ n'apparaît plus d'une fois dans `l`. Par exemple, $\texttt{sum}(x_0, x_1)$ est linéaire mais pas $\texttt{eq}(x_0, x_0)$.

Question à développer pendant l'oral 2 *Donner la complexité en temps de* `find_match t l` *en fonction de la taille de* `t` *et* `l` *dans les cas suivants :*
- *le terme* `l` *est linéaire.*
- *dans le cas général, sans hypothèse sur* `l`.

1.2 Système de réécriture

Un système de réécriture $\mathcal{R}$ est un ensemble fini de règles permettant de réécrire dans un terme. Une règle de réécriture R est de la forme $l \Rightarrow r$, où l et r sont des termes de règle. Cette règle signifie que l'on peut réécrire l en r dans **n'importe quel sous-terme** d'un terme. Par exemple, la règle R_0 suivante (représentée à gauche avec des mots, et à droite avec des arbres) :

$$\texttt{sum}(\mathsf{x}_0, \texttt{s}(\mathsf{x}_1)) \Rightarrow \texttt{sum}(\texttt{s}(\mathsf{x}_0), \mathsf{x}_1)$$



(la règle R_0)

utilise deux variables x_0 et x_1. En OCaml, on représentera une règle $l \Rightarrow r$ par le couple (l, r), à l'aide du type suivant :

```
type rule = rule_term * rule_term
```

On écrit $t \to_{\mathsf{R}} t'$ quand le terme t peut se réécrire en t' par la règle R. Par exemple, on a :

$$\texttt{sum}(\texttt{s}(\texttt{z}), \texttt{s}(\texttt{z})) \to_{\mathsf{R}_0} \texttt{sum}(\texttt{s}(\texttt{s}(\texttt{z})), \texttt{z})$$

Une règle de réécriture peut s'appliquer en profondeur, et plusieurs règles peuvent s'appliquer au même terme. Ainsi, en notant R_1 la règle $\texttt{eq}(\mathsf{x}_0, \mathsf{x}_0) \Rightarrow \texttt{true}$, on a (on souligne le sous-terme où la réécriture a lieu) :

$$\begin{aligned}
\texttt{pair}\big(\underline{\texttt{eq}(\texttt{z},\texttt{z})},\ \texttt{eq}(\texttt{z},\texttt{z})\big) &\to_{\mathsf{R}_1} \texttt{pair}\big(\underline{\texttt{true}}\ ,\ \texttt{eq}(\texttt{z},\texttt{z})\big) \\
\texttt{pair}\big(\texttt{eq}(\texttt{z},\texttt{z}),\ \underline{\texttt{eq}(\texttt{z},\texttt{z})}\big) &\to_{\mathsf{R}_1} \texttt{pair}\big(\texttt{eq}(\texttt{z},\texttt{z}),\ \underline{\texttt{true}}\ \big)
\end{aligned}$$

Plus généralement, détaillons les étapes de la réécriture par une règle $l \Rightarrow r$, dans le cas particulier de l'exemple ci-dessus à gauche :

- On identifie le sous-terme s où l'on applique la réécriture : ici, s est le sous-terme $\texttt{eq}(\texttt{z}, \texttt{z})$ de gauche.
- On *matche* l avec le sous-terme s pour obtenir une substitution σ telle que s soit égal à $\sigma(l)$. Ici, on prend $\sigma = \{\mathsf{x}_0 \mapsto \texttt{z}\}$.
- On remplace le sous-terme s par $\sigma(r)$.

Question 4 *Implémenter la fonction de signature :*

```
val rewrite1 : term -> rule list -> term option
```

telle que `rewrite1 t rules` *réécrive une des règles de* `rules` *dans* `t`*. Si aucune règle de* `rules` *ne s'applique,* `rewrite1` *renverra* `None`*.*
On fera la première réécriture possible sur un sous-terme de `t` *dans une recherche en profondeur, c'est-à-dire que l'on parcourra* `t` *en profondeur jusqu'à atteindre le premier sous-terme auquel l'une des règles s'applique. Si plusieurs règles s'appliquent à ce sous-terme, on utilisera celle qui apparaît en premier dans* `rules` *lors d'un parcours de la liste.*

Notons `terms = read_terms "data/U0/tq4.txt"` *et* `rules = read_rules "data/U0/rq4.txt"`*. Soit* `t` *le k-ième terme de* `terms`*. Si* `rewrite1 t rules = Some t'`*, alors noter* `Some (hash t')` *sur la fiche réponse (sinon, noter* `None`*), pour les valeurs de k suivantes :*

a) $k = 0$ ***b)*** $k = 1$ ***c)*** $k = 2$ ***d)*** $k = 3$

On généralise la notion de linéarité comme suit : on dit qu'un terme de règle `l` est L-linéaire si L est le plus petit entier tel qu'aucune variable n'apparaisse plus de L fois dans `l`. Par exemple, $\mathtt{f}(\mathtt{x}_0, \mathtt{x}_0, \mathtt{x}_1)$ est 2-linéaire car $\mathtt{x}_0$ apparaît deux fois.

Question à développer pendant l'oral 3 *Donner la complexité en temps et borner la taille de la sortie de* `rewrite1 t rules`*. On exprimera ces deux quantités en fonction, entre autres, des paramètres suivants :*

- *le nombre N de règles dans* `rules`*, et le nombre V de variables différentes dans les règles ;*
- *une borne L sur la L-linéarité des termes de règles apparaissant à droites des règles de* `rules`*, c.-à-d. que pour chaque règle* `l` $\Rightarrow$ `r` *de* `rules`*, le terme* `r` *est L'-linéaire pour $L' \leq L$.*

Question à développer pendant l'oral 4 *Simplifier les expressions des complexités en temps et des bornes sur la taille de la sortie précédentes dans le cas où l'ensemble des règles* `rules` *est supposé fixé.*

Une séquence de réécritures au départ d'un terme t pour un ensemble de règles $\mathcal{R}$ est une suite de termes $t_0, \ldots, t_l$ telle que $t_0 = t$ et $t_0 \to_{\mathcal{R}} t_1 \to_{\mathcal{R}} \ldots \to_{\mathcal{R}} t_l$. La longueur d'une séquence est le nombre de réécritures dans celle-ci : ainsi, la séquence précédente est de longueur l. Un terme t est dit irréductible pour $\mathcal{R}$ si aucune règle de $\mathcal{R}$ ne s'applique à t, c-.à.d. si il n'existe pas de terme t' tel que $t \to_{\mathcal{R}} t'$.

Question 5 *Implémenter la fonction de signature :*

```
val rewriteN : term -> rule list -> term * int
```

telle que `rewriteN t rules` *applique la fonction* `rewrite1` *de façon répéter pour calculer une séquence de réécritures au départ de* `t` *par* `rules` *jusqu'à obtenir un terme* `t'` ***irréductible****.*
La fonction `rewriteN t rules` *renvoie le couple* `(t', l)` *composé du terme finale* `t'` *et de la longueur* `l` *de la séquence de réécritures calculée. Plus précisément, on a* `t'` *irréductible tel que*

$$\mathtt{t} = \mathtt{t}_0 \to_{\mathcal{R}} \ldots \to_{\mathcal{R}} \mathtt{t}_l = \mathtt{t'} \quad \text{avec} \quad (\mathtt{rewrite1}\ \mathtt{t}_i\ \mathtt{rules} = \mathtt{t}_{i+1}) \quad \text{pour tout } 0 \leq i < l.$$

On admet que ce processus termine, c.-à.d. que `t'` et `l` sont bien définis sur les entrées données par l'énoncé.

Notons `terms = read_terms "data/U0/tq5.txt"` *et* `rules = read_rules "data/U0/rq5.txt"`*. Soit* `t` *le k-ième terme de* `terms`*. Calculer l'entier* `snd (rewriteN t rules)` *pour les valeurs de k suivantes :*

a) $k = 0$ ***b)*** $k = 1$ ***c)*** $k = 2$ ***d)*** $k = 3$

Question à développer pendant l'oral 5 *Donner la complexité en temps et borner la taille de la sortie de* `rewriteN t rules` *en supposant l'ensemble des règles fixé. On ne fera aucune hypothèse de linéarité sur les règles utilisées.*

2 Partie 2 (C) : Table de hachage

Vous devez utiliser C tout au long de cette partie.

Code fourni Le fichier `base.c` contient les éléments de code suivant :

— un ensemble d'inclusions usuelles en en-tête (`<stdlib.h>`, `<stdio.h>`, ...)

— une fonction de génération de nombres pseudo-aléatoires `rands` et une fonction de hachage `hash` qui sont décrites plus bas.

Implémentation d'un dictionnaire Un dictionnaire est une structure de données abstraite permettant de stocker un ensemble de liaisons de clé/valeur `(k,v)` telle que chaque clé `k` soit unique. Typiquement, il est attendu d'une structure de dictionnaire qu'elle supporte les opérations suivantes de façon efficace :

— `search(d,k)` : cherche la valeur `v` associée à la clé `k` dans le dictionnaire `d`.

— `add(d,k,v)` : ajoute la liaison clé/valeur `(k,v)` dans le dictionnaire `d`. Si une liaison `(k,v0)` était déjà présente pour la clé `k`, elle est remplacée par la nouvelle liaison `(k,v)`.

— `remove(d,k)` : supprime la liaison de clé `k` dans le dictionnaire `d`.

Nous allons implémenter un dictionnaire à l'aide d'une table de hachage par sondage linéaire. Nous considérerons des clés et valeurs de type `uint32_t`, et utiliserons la fonction de hachage fournie. Celle-ci est de signature `uint32_t hash(uint32_t key, uint32_t N)`, permet de hacher une clé `key` pour le paramètre `N` et est telle que `hash(key, N)` $\in \{0;\ldots;N-1\}$ pour tout `key, N`.

Le hachage par sondage linéaire fonctionne comme suit. On va stocker un ensemble de `n` liaisons clé/valeur `(k1,v1), ..., (kn,vn)` dans un tableau de taille `N` fixée. Dans un premier temps, on supposera le tableau suffisamment grand pour pouvoir stocker toutes les liaisons, c'est-à-dire qu'on demande que `n` $\leq$ `N`. En temps normal, toutes les cellules du tableau ne sont pas remplies, et on aura donc `n` $<$ `N`.
Par défaut, une liaison `(k,v)` est stockée dans la case d'indice `hash(k,N)` du tableau. Par exemple :

```
N = 6
Liaisons : (k1,v1), (k2,v2)
 hash(k1,N) = 5
 hash(k2,N) = 1
```

indice	hash	clé	valeur
0			
1	1	k2	v2
2			
3			
4			
5	5	k1	v1

Cependant, lors de l'ajout d'une liaison `(k,v)`, il est possible qu'une valeur soit déjà présente à la case d'indice `hash(k,N)` (on dit qu'il y a une collision). Dans ce cas, on stocke la liaison `(k,v)` dans la première case libre disponible à partir de l'indice `hash(k,N)`. Par exemple :

```
N = 6
Liaisons : (k1,v1), (k2,v2), (k3,v3), (k4,v4)
 hash(k1,N) = 5
 hash(k2,N) = 1
 hash(k3,N) = 1 /* collision (k2), déplacée indice 2 */
 hash(k4,N) = 2 /* collision (k3), déplacée indice 3 */
```

indice	hash	clé	valeur
0			
1	1	k2	v2
2	1	k3	v3
3	2	k4	v4
4			
5	5	k1	v1

Les indices du tableau doivent être considéré modulo sa taille `N`. Par exemple, l'ajout d'une liaison `(k,v)` de hache `5` dans le tableau ci-dessus insérera celle-ci dans la case suivant la case d'indice `5` (car celle-ci est occupée), c'est à dire dans la case d'indice `0`. Enfin, on notera que le tableau obtenu peut dépendre de l'ordre d'ajout des liaisons dans celui-ci.

On appelle surcharge d'une table de hachage par sondage linéaire `d` de taille `N`, et l'on note `surcharge(d)`, le nombre de liaisons `(k,v)` stockées dans la table à une autre position que `hash(k,N)`. Ainsi, la table ci-dessus a une surcharge de 2, à cause des liaisons `(k3,v3)` et `(k4,v4)`.

Question 6 *Implémenter une structure de donnée réalisant une table de hachage par sondage linéaire, l'opération d'ajout* `add(d,k,v)`*, et la fonction* `surcharge(d)`*.*
(On ne demande pas d'implémenter les fonctions `search` *et* `remove`*.)*
Créer une table de hachage `d` *de taille* `N`*, et utiliser la fonction* `rands` *qui vous est fournie pour calculer la surcharge de la table de hachage* `d` *calculée par le programme suivant :*

```
uint32_t *r = rands(u0,len);

for (int i = 0; i < len; i++){
  uint32_t key = r[i] % (10 * len);
  uint32_t val = (17 * (r[i] + i)) % 20;
  add(d,key,val);
}
```

pour `u0` $= u_0$ *et* `N` $= 4 \times$ `len`*, et cela pour les valeurs de* `len` *suivantes :*

a) `len` $= 100$ ***b)*** `len` $= 10\,000$ ***c)*** `len` $= 1\,000\,000$ ***d)*** `len` $= 10\,000\,000$

Question à développer pendant l'oral 6 *Décrire votre choix de structure de données. Expliquer informellement comment évolue la complexité en temps de l'ajout d'un élément dans la table en fonction du nombre de liaisons déjà présentes dans celle-ci.*

Question à développer pendant l'oral 7 *Décrire l'algorithme que vous utiliseriez pour implémenter les fonctions* `search` *et* `remove` *décrites au début de cette partie (on ne demande pas de réellement les implémenter).*

Le taux de remplissage τ d'une table de hachage est le ratio du nombre de cellules utilisées sur le nombre de cellules total de la table. Lorsque celui-ci devient trop grand, il est nécessaire d'augmenter la taille de la table de hachage en passant d'une table de taille `N` à une table de taille `2 * N` : on dit qu'on re-dimensionne la table. Concrètement, si `d` est une table de taille `N`, on crée une nouvelle table de taille `2 * N`, dans laquelle on transfère tous les éléments qui étaient présents dans `d`. On fera ce transfert élément par élément, en parcourant l'ancienne table dans l'ordre des indices croissants. On note `resize(d)` cette opération. Bien sûr, lors de l'ajout à la nouvelle table de taille `2*N`, les éléments sont de nouveau hachés, à l'aide la fonction `hash(., 2*N)` et non plus avec `hash(., N)` comme précédemment.
Une stratégie consiste à re-dimensionner la table à chaque fois que son taux de remplissage dépasse strictement 25%. Par exemple, si $N = 16$ et que 4 liaisons sont déjà dans la table, celle-ci sera re-dimensionnée lors de l'ajout d'une 5^{e} liaison. On note `add_resize(d,k,v)` la fonction qui se comporte comme `add(d,k,v)`, à ceci près qu'elle re-dimensionne la table `d` si nécessaire après l'ajout de la liaison `(k,v)`.

Question 7 *Implémenter l'opération* `add_resize(d,k,v)`*.*
Créer une table de hachage `d` *de taille initiale* `16`*, et utiliser la fonction* `rands` *fournie pour calculer la surcharge de la table de hachage* `d` *calculée par le programme suivant :*

```
uint32_t *r = rands(u0,len);

for (int i = 0; i < len; i++){
  uint32_t key = r[i] % (10 * len);
  uint32_t val = (17 * (r[i] + i)) % 20;
  add_resize(d,key,val);
}
```

pour `u0` $= u_0$ *et* $N = 4 \times$ `len`*, et cela pour les valeurs* `len` *suivantes :*

a) `len` $= 100$ ***b)*** `len` $= 10\,000$ ***c)*** `len` $= 1\,000\,000$ ***d)*** `len` $= 10\,000\,000$

3 Partie 3 (OCaml) : Réécriture efficace

Vous devez utiliser OCaml tout au long de cette partie.

Code fourni Cette partie suit la Partie 1, et utilisera le même fichier de base `base.ml`.

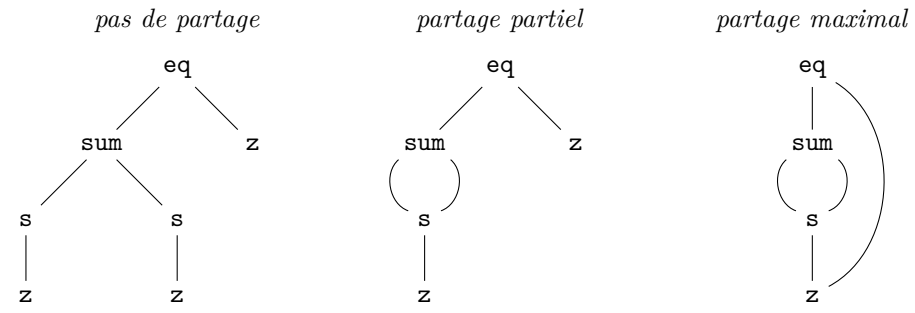
Terme abstrait et représentation Dans cette partie, on fera la différence entre un terme abstrait et sa représentation par un terme OCaml. Par exemple, le terme abstrait `z` peut être représenté par le terme OCaml `Tfun(Tag.default,"z",[])`. Un même terme abstrait peut avoir plusieurs représentations par des termes OCaml, de façon plus ou moins efficace en mémoire. Par exemple, voici deux représentations du terme `eq`$(\mathtt{z},\mathtt{z})$:

```
let d = Tag.default in                  let d = Tag.default in
Tfun(d,"eq",[Tfun(d,"z",[]);            let z = Tfun(d,"z",[]) in
             Tfun(d,"z",[])])           Tfun(d,"eq",[z; z])
```

La représentation de droite utilise un espace mémoire réduit car le sous-terme `z` est partagé : en effet, celui-ci n'est construit qu'une seule fois à droite par la liaison `let z = ...`.

3.1 Partage maximal

On dit qu'un terme OCaml `t` de type `term` est *hash-consé* s'il a la propriété de partage maximal, c.-à-d. si dès que deux sous-termes de `t` sont égaux, ils sont représentés **physiquement dans la mémoire** par le même objet OCaml. Par exemple, voilà trois représentations possibles (sous forme de graphes) du même terme abstrait `eq`$(\mathtt{sum}(\mathtt{s}(\mathtt{z}),\mathtt{s}(\mathtt{z})),\mathtt{z})$:



Nous allons utiliser la technique suivante pour construire des termes OCaml *hash-consés*.

Tags Chaque terme *hash-consé* `t = Tfun(tag,s,args)` aura un tag `tag` (différent de `Tag.default`). Le tag d'un terme caractérisera de façon unique celui-ci :

— les termes *hash-consés* représentant des termes abstraits différents auront des tags différents ;

— les termes OCaml représentant le même terme abstrait seront physiquement égaux, et auront donc (entre autres) le même tag.

Hash-consing On maintient l'ensemble des termes *hash-consés* déjà construits dans une structure de données $\mathcal{H}$ (détaillée par la suite). Pour construire le terme *hash-consé* `f(args)`, où les sous-termes `args` sont déjà *hash-consés*, on cherche si `f(args)` est déjà présent dans $\mathcal{H}$: si c'est le cas, on réutilise le terme OCaml dans $\mathcal{H}$; sinon, on créé un nouveau tag `tag` frais à l'aide de `Tag.create()`, on stocke `Tfun(tag,f,args)` dans $\mathcal{H}$ et on renvoie ce nouveau terme.

Structure $\mathcal{H}$ La structure de donnée $\mathcal{H}$ et la fonction `fun_hashcons` permettant de la manipuler sont décrites ci-dessous. Celles-ci sont **déjà fournies** dans le module `HashCons` du fichier `base.ml`. **Il ne faut pas les ré-implémenter ou les recopier !**

On implémente la structure $\mathcal{H}$ par une table de hachage ayant le type enregistrement suivant :

```
type hashtbl : { table : term list array; size : int }
```

où `size` est la taille du tableau `table`, et la cellule en position `h` du tableau `table` contient la liste `l` des termes *hash-consés* de hache `h`, où ce hache est obtenu à l'aide de la fonction fournie `hash_for_hashconsing`.

Enfin, le module `HashCons` définit une seule fonction de signature :

```
val fun_hashcons : string -> term list -> term
```

telle que `(fun_hashcons s args)` calcule le terme *hash-consé* `s(args)`, en supposant que les termes `args` sont **déjà hash-consés**.

Question à développer pendant l'oral 8 *Regarder le code du module* `HashCons` *et répondre aux questions suivantes :*

— *Quel est l'intérêt d'utiliser la fonction de hachage* `hash_for_hashconsing` *plutôt que la fonction de hachage* `hash` *de la Question 1 ?*

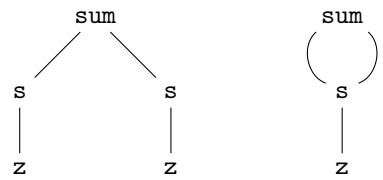
— *Pourquoi utiliser la fonction* `equal_for_hashconsing` *plutôt que la fonction d'égalité usuelle* `=` *? Pourquoi cette fonction est-elle correcte ?*

— *Comment la fonction* `fun_hashcons` *réalise-t-elle sa spécification ?*

Soit `t` un terme OCaml. La taille de `t`, notée $\mathsf{taille}(t)$, est la taille du terme abstrait représenté par `t`, c.-à-d. :

$$\mathsf{taille}(f(t_1 \ldots, t_n) = 1 + \sum_{i=1}^{n} \mathsf{taille}(t_i)$$

La taille en mémoire de `t` est la taille de la représentation en mémoire de `t`, c.-à-d. le nombre de nœuds du graphe représentant `t`. Par exemple, les deux représentations suivantes :



du même terme abstrait $t = \texttt{sum}(\texttt{s}(\texttt{z}), \texttt{s}(\texttt{z}))$ sont toutes deux de taille 5, mais celle de gauche est de taille en mémoire 5, alors celle de droite est de taille en mémoire 3.

Question 8 *Implémenter la fonction* `hashcons` *de type* `term -> term` *telle que* `hashcons t` *calcule le hash-consé de* `t`*.*

Implémenter les fonctions `size` *et* `size_mem` *de type* `term -> int` *telles que* `size t` *calcule la taille du terme* `t`*, et* `size_mem t` *calcule la taille en mémoire d'un terme* `t` *(que l'on supposera* ***hash-consé****).*

Notons `terms = read_terms "data/U0/q8.txt"`*.*
Soit `t` *le k-ième terme de* `terms`*. Calculer le couple* `(size t', size_mem t')`*, où* `t' = hashcons t` *pour les valeurs de k suivantes :*

a) $k = 0$ ***b)*** $k = 1$ ***c)*** $k = 2$ ***d)*** $k = 3$

Question à développer pendant l'oral 9 *Détailler votre implémentation de* `size_mem` *et donner sa complexité en temps.*

3.2 Réécriture et partage maximal

Question 9 *Réécrire la fonction* `rewriteN` *en supposant que le terme de départ est hash-consé, et en garantissant que le terme calculé l'est aussi. On appellera cette nouvelle implémentation* `rewriteN_hashconsed`*.*

Indications. *Copier-coller votre code précédent, modifier attentivement celui-ci pour garantir que des termes hash-consés sont produits, et optimiser le code en exploitant le hash-consing.*

Attention, une implémentation trop naïve de la fonction `size_mem` *ne sera pas capable de gérer les plus grands termes construits dans cette question.*

Notons `terms = read_terms "data/U0/tq9.txt"` *et* `rules = read_rules "data/U0/rq9.txt"`*.*
Soit `t` *le k-ième terme de* `terms`*. Calculer le couple* `(size_mem t', len)` *telle que* `rewriteN_hashconsed t rules = (t', len)` *pour les valeurs de k suivantes :*

a) $k = 0$ ***b)*** $k = 1$ ***c)*** $k = 2$ ***d)*** $k = 3$

Question à développer pendant l'oral 10 *Donner la complexité en temps et borner la taille de la sortie de* `rewriteN_hashconsed t rules n` *en supposant l'ensemble des règles fixé, et que les termes de règles à droites des règles sont tous au plus L-linéaire.*

On rappelle que la longueur d'une séquence est le nombre de réécritures dans celle-ci.

Question 10 *Implémenter la fonction de signature :*

```
val longest : term -> rule list -> int
```

telle que `longest t rules` *calcule la longueur de la* ***plus longue*** *séquence de réécritures au départ de* `t` *par* `rules`*. On considérera toutes les séquences de réécritures possibles, et pas seulement celles obtenues par* `rewrite1` *: en particulier, l'ordre des règles dans* `rules` *n'a plus d'importance, et on ne s'arrêtera pas à la première réécriture rencontrée lors d'un parcours en profondeur. On supposera que toutes les séquences de réécritures au départ de* `t` *sont de longueur finie.*

Notons `terms = read_terms "data/U0/tq10.txt"` *et soit* `t` *le k-ième terme de* `terms`*.*
Notons `rules = read_rules "data/U0/rq10.txt"`*.*
Calculer `longest t rules` *pour les valeurs de k suivantes :*

a) $k = 0$ ***b)*** $k = 1$ ***c)*** $k = 2$ ***d)*** $k = 3$

Question à développer pendant l'oral 11 *Décrire votre implémentation de la fonction* `longest`*. Exprimer sa complexité en temps en fonction, entre autres, de la longueur L de la plus longue séquence de réécritures au départ de* `t`*. On supposera l'ensemble des règles fixé.*

Fiche réponse type : Réécriture et partage

$\widetilde{u_0}$: 1

Question 1

a) 73085

b) 79891

c) 32362

d) 562

Question 2

a) 64609

b) 21394

c) 81884

d) 15741

Question 3

a) Some 128585

b) None

c) Some 49680

d) None

Question 4

a) Some 79136

b) Some 16665

c) Some 820

d) None

Question 5

a) 2

b) 6

c) 52

d) 1679

Question 6

a) 9

b) 1092

c) 112930

d) 1130432

Question 7

a) 4

b) 755

c) 108190

d) 667827

Question 8

a) (8,6)

b) (45,24)

c) (195,82)

d) (695,282)

Question 9

a) (8,22)

b) (10,37)

c) (135,1228)

d) (263,2572)

Question 10

a) 5

b) 7

c) 18

d) 22

Fiche réponse : Réécriture et partage

Nom, prénom, u_0 : .....

Question 1

a)

b)

c)

d)

Question 2

a)

b)

c)

d)

Question 3

a)

b)

c)

d)

Question 4

a)

b)

c)

d)

Question 5

a)

b)

c)

d)

Question 6

a)

b)

c)

d)

Question 7

a)

b)

c)

d)

Question 8

a)

b)

c)

d)

Question 9

a)

b)

c)

d)

Question 10

a)

b)

c)

d)