Devoir Non Surveillé 4

À rendre le 22 avril 2024

INFORMATIQUE MP2I

Tournez la page S.V.P.

1 Expressions arithmétiques

On définit en OCaml les expressions littérales comme des arbres binaires, où les nœuds internes sont étiquetés par des opérateurs binaires (ici + et ×), et les feuilles sont étiquettées soit par un nombre entier soit par la variable x :

Expressions littérales

```
type operator = Add | Mul
type expr =
  | Int of int
  | Var
  | Op of expr * operator * expr

let exemple = Op (Int 2, Add, Op (Op (Int 3, Add, Int 7), Mul, Int 4))
let exemple_bis = Op (Op (Int 3, Mul, Var), Add, Int (-126))
```

☐ 1 – Écrire une fonction nombres : expr –> int qui compte le nombre de feuilles (nombres ou variables) dans une expression littérale.

☐ 2 – Écrire une fonction operateurs : expr –> int qui compte le nombre d'opérateurs binaires dans une expression littérale.

☐ 3 – Que peut-on dire des résultats des deux fonctions précédentes sur une même expression littérale ? le démontrer.

☐ 4 – Écrire une fonction eval : int –> expr –> int qui prend en entrée un entier x et une expression littérale et qui renvoie le résultat de l'expression si la variable vaut x.
On pourra vérifier les valeurs de eval 0 exemple et eval 42 exemple_bis.

☐ 5 – Écrire des fonctions prefixe : expr –> string,
postfixe : expr –> string et infixe : expr –> string qui permettent de transformer une expression littérale en une chaîne de caractères qui la représente en notation préfixe, postfixe ou infixe.

☐ 6 – En réalité, les expressions que l'on manipule ici sont des polynômes (il n'y a pas de division). Écrire une fonction degree : expr –> int qui calcule une borne sur le degré d'un polynôme représenté par une expression littérale.
Donner la complexité de votre fonction.
On pourra simplifier les disjonctions de cas en utilisant 0 *comme borne sur le degré du polynôme identiquement nul* $P(X) = 0$.

☐ 7 – Écrire une fonction egales : expr –> expr –> bool qui teste l'égalité entre deux expressions littérales basée sur l'estimation du degré implémentée à la fonction précédente.
Estimer la complexité de votre fonction.

Nous souhaitons à présent développer et réduire nos expressions pour obtenir une forme canonique. Nous allons donc représenter à présent les polynômes par une suite finie de monômes. Remarquons que le polynôme identiquement nul sera représenté par une liste vide, les monômes étant donc de degré positif ou nul. Pour que la représentation canonique soit unique, nous trierons les monômes par degré strictement décroissant.

Polynômes

```
type monomial = {deg : int; coeff : int}
type polynomial = monomial list
```

☐ 8 – Écrire une fonction add_monomial : monomial –> polynomial –> polynomial qui ajoute un monôme à un polynôme.
En déduire une fonction sum : polynomial –> polynomial –> polynomial qui somme deux polynômes.

☐ 9 – Écrire une fonction distribute : monomial –> polynomial –> polynomial qui multiplie un polynôme par un monôme donné en utilisant la distributivité.
En déduire une fonction prod : polynomial –> polynomial –> polynomial qui multiplie deux polynômes.

☐ 10 – Écrire une fonction expand_simplify : expr –> polynomial qui développe et réduit une expression littérale pour en déduire sa forme polynomiale.

☐ 11 – Expliquer comment calculer le degré d'un poylnôme donné sous forme de liste de monôme, ainsi que comment tester l'égalité de deux polynômes.
Estimer les complexités de ces deux opérations.

2 À cheval sur deux tableaux triés

Le but de cette partie est de déterminer le $k+1$-ième plus petit élément, dans deux tableaux triés.
Par exemple, si on considère les tableaux , 2, 4, 6, 80, 2, 4, 6, 8 et , 3, 5, 7, 91, 3, 5, 7, 9, le 4-ième plus petit élément est 3. C'est-à-dire que si on regroupait les deux tableau en un tableau trié, il aurait la position 3. On ne vérifiera pas que les tableaux donnés en entrée sont effectivement triés. On suppose dans toute la suite que $k < n_1 + n_2$, sinon le problème n'a pas de solution.

☐ 12 – Écrire une fonction **int** ∗fusion(**int** tab1[], **int** tab2[], **int** n1, **int** n2) qui renvoie un tableau trié de taille n_1+n_2 contenant les éléments de t_1 et t_2, où n_i est la longueur de t_i. On prendra soin d'allouer la mémoire dynamiquement (malloc).

☐ 13 – En déduire une fonction **int** select_naif(**int** k, **int** tab1[], **int** tab2[], **int** n1, **int** n2) qui détermine le $k+1$-ième plus petit élément de $t_1 \cup t_2$ en utilisant fusion. On prendra soin de libérer la mémoire allouée par fusion.

Pour éviter l'allocation et l'initialisation de ce tableau, on se propose d'utiliser le code suivant, manipulant le même tableau, mais sans le créer.

```
int select_lin(int k, int tab1[], int tab2[], int n1, int n2){
  assert(k < n1 + n2);                          // 1
  int i = 0, j = 0;                             // 2
  while (i + j < k) {                           // 3
    if (j >= n2 || tab1[i] < tab2[j])           // 4
      { i = i + 1; }                            // 5
    if (i >= n1 || tab1[i] >= tab2[j])          // 6
      { j = j + 1; }                            // 7
  }
  if (j == n2 || tab1[i] < tab2[j])             // 8
    { return tab1[i]; }                         // 9
  else { return tab2[j]; }                      // 10
}
```

☐ 14 – Dessiner le graphe de flot de contrôle de la fonction select_lin et proposer un jeu de test couvrant l'ensemble de ses branches.

☐ 15 – Donner un variant permettant de montrer la terminaison de la boucle **while** et donc de la fonction select_lin. En déduire la complexité temporelle dans le pire des cas en utilisant la notation $O(\cdot)$.

☐ 16 – Démontrer à l'aide d'un invariant de la boucle **while** la correction partielle de la fonction select_lin. On rappelle que les tableaux t_1 et t_2 sont supposés triés.

On souhaite désormais trouver un algorithme plus efficace, utilisant mieux le fait que les tableaux soient triés. Soit i, tel que $0 \leq i$, $i < n_1$, $i \leq k$. Posons $j = k - i$ (remarquons que $j < n_2$ car $k < n_1 + n_2$).

☐ 17 – Montrer que si $t_1[i-1] \leq t_2[j]$ et $t_2[j-1] \leq t_1[i]$, alors le $(k+1)$-ième plus petit élément est $\min(t_1[i], t_2[j])$.

☐ 18 – Montrer que si $t_1[i-1] > t_2[j]$, alors le $(k+1)$-ième plus petit élément est plus petit que $t_1[i]$.

☐ 19 – Déduire des deux questions précédentes une fonction **int** select_log(**int** k, **int** tab1[], **int** tab2[], **int** n1, **int** n2) qui détermine le $(k+1)$-ième plus petit élément en temps $O(\log(n_1))$.

☐ 20 – Expliquer comment on pourrait résoudre le problème dans le pire des cas en $O(\log(\min(n_1, n_2)))$.

3 Scrabble – OCaml (20 pts)

Dans cette section, nous allons nous intéresser à un programme qui joue au Scrabble.
Chaque lettre de l'alphabet latin ('A','B',...,'Z') est associé à un nombre de points allant de 1 à 10.
Étant donné un ensemble de 7 lettres, on cherche à faire le mot valide valant le plus de points. Si on arrive à placer ses 7 lettres, on gagne un bonus de 50 points.
Nous allons ici ignorer le placement du mot sur le plateau.
Pour savoir si un mot est valide, nous disposons d'un fichier `dictionnaire.txt` qui contient la liste de tous les mots valides, séparés par des espaces (on suppose la présence d'un espace supplémentaire en première et dernière position, de telle manière que chaque mot est entouré par deux espaces). Nous affectons son contenu à une variable dict : string, dont on peut accéder au i-ième caractère par dict.[i].

```
let f = open_in "dictionnaire.txt"
let dict = input_line f
let () = close_in f
```

Dans un premier temps, intéressons nous à la possibilité de faire un Scrabble, c'est-à-dire de placer les 7 lettres. Il faut donc chercher dans le dictionnaire tous les mots contenant nos 7 lettres.

☐ 21 – Dans le pire des cas, combien peut-on former de mots (pas nécessairement valides) différents que l'on doit tester avec 7 lettres ?

Nous allons adapter l'algorithme de Rabin-Karp pour trouver aussi les anagrammes d'un motif, c'est-à-dire un facteur du mot qui contient les mêmes lettres que le motif, mais pas nécessairement dans le même ordre. Pour ceci, on propose de donner un code à chaque caractère ainsi : '␣' -> 0, 'A' -> 1, 'B' -> 2...'Z' -> 26.

```
let code c =
  if c = ' ' then 0 else
    begin
      assert ('A' <= c && c <= 'Z');
      int_of_char c - int_of_char 'A' + 1
    end
```

On propose d'utiliser la fonction de hachage suivante sur les mots de taille arbitraire :

$$h : c_0, \ldots, c_n \mapsto \prod_{i=0}^{n} p_{\bar{c}_i},$$

où p_k désigne le k-ième nombre premier, et $\bar{c}$ désigne le code du caractère c.

☐ 22 – Montrer que deux mots $\mathbf{b}$ et $\mathbf{c}$ sont des anagrammes si et seulement si $h(\mathbf{b}) = h(\mathbf{c})$.

On suppose à présent qu'on dispose d'un tableau $\mathbf{p}$ contenant les 27 premiers nombres premiers :

```
let p = [|
     2; 3; 5; 7;11;
     13;17;19;23;29;
     31;37;41;43;47;
     53;59;61;67;71;
     73;79;83;89;97;
     101;103|]
```

☐ 23 – Écrire une fonction hash : string -> int -> int qui prend en entrée une chaîne de caractère $\mathbf{c}$ et un entier n et qui calcule $h(c_0, \ldots, c_{n-1})$.
Cette fonction vérifiera que la taille de $\mathbf{c}$ est au moins n.

☐ 24 – Si on trouve un facteur $\mathbf{b} = b_0 \ldots b_6$ du dictionnaire tel que $h(\mathbf{b}) = h(\mathbf{c})$, peut-on être sur de l'existence d'un anagramme de $\mathbf{c}$ dans le dictionnaire ? Justifier.

☐ 25 – Paul pense que vérifier l'égalité des images par h n'est pas suffisant et propose de tester que le motif contienne bien deux espaces en considérant les anagrammes de $\mathbf{c}' = \text{␣}c_0 \dots c_6\text{␣}$. Si on trouve un facteur $\mathbf{b} = b_0 \dots b_8$ du dictionnaire tel que $h(\mathbf{b}) = h(\mathbf{c}')$, peut-on être sur de l'existence d'un anagramme de $\mathbf{c}$ dans le dictionnaire ? Justifier.

☐ 26 – Écrire une fonction verifie : 'a -> 'a -> string -> int -> int -> bool qui prend en argument la valeur $h(\mathbf{c}')$ pour un certain motif $\mathbf{c}' = \text{␣}\mathbf{c}\text{␣}$, avec $\mathbf{c}$ de taille n, la valeur $h(b_i, b_{i+n+1})$, le dictionnaire $\mathbf{b}$, la taille du motif n et la position i et qui vérifie si on a bien un anagramme de $\mathbf{c}$ présent en position $i+1$ dans le dictionnaire $\mathbf{b}$.
remarque : la fonction ne prend pas le motif $\mathbf{c}$ *en argument.*

☐ 27 – Peut-on avoir un dépassement de capacité lorsque l'on calcule une image par h d'un facteur de taille 9 ? Pour la suite, on supposera que l'on a jamais de dépassement de capacité.

On propose l'implémentation suivante à compléter d'une variante de l'algorithme de Rabin-Karp :

variante de Rabin-Karp

```
let rabin_karp motif texte =
  let n = String.length motif in
  let h = hash motif n in
  let emp = hash texte n in
  let rec parcours emp i =
    if i > String.length texte - n then
      false
    else if verifie h emp texte n i then
      true
    else
      let nouv_emp = (* A COMPLETER *) in
      parcours nouv_emp (i+1)
  in
  parcours emp 0
```

☐ 28 – Compléter l'algorithme en remplaçant le commentaire.

☐ 29 – Quelle est la complexité de cet algorithme ?

☐ 30 – Peut-on adapter cette technique pour chercher aussi les anagrammes de sous-ensembles de 6 lettres de notre tirage ? Est-ce avantageux ?

4 Complexité en moyenne des tables de hachages

Dans cette section, nous estimons la complexité moyenne dans le pire des cas pour les opérations de recherche et d'insertion dans une table de hachage. Nous supposerons que la fonction de hachage s'évalue en $O(1)$.
Nous notons n le nombre d'éléments dans la table de hachage et m le nombre de cases du tableau. Nous noterons $\alpha = \frac{n}{m}$ le facteur de remplissage de la table.
Nous supposons que pour toute entrée x, notre fonction de hachage renvoie un nombre aléatoire entre 0 inclu et m exclu, et que les résultats pour deux entrées distinctes x et y sont indépendants.
Nous avons donc que la probabilité de collision pour deux éléments donnés est $\frac{1}{m}$.

☐ 31 – Quelle est la probabilité qu'il y ait au moins une collision parmi les n premiers éléments qui ont été insérés en fonction de n et m ?

4.1 Gestion des collisions par chaînage

Dans cette section, on décide de stocker dans chaque case du tableau une liste chaînée d'associations pour gérer les collisions. Ainsi, la complexité de la recherche est proportionnelle à la taille de la liste contenue dans la case explorée. La complexité de l'addition d'un élément est $O(1)$ si on fait les ajouts en tête de liste.

☐ 32 – Après n insertions dans des cases aléatoires parmi m cases, quelle est l'espérance du nombre d'éléments dans une case donnée ? En déduire la complexité en espérance dans le pire des cas d'une recherche.

4.2 Gestion des collisions par adressage ouvert avec sondage linéaire

Dans cette section, on décide pour gérer les collisions que si la case est déjà prise, on va chercher dans la case suivante, jusqu'à trouver une case vide.

☐ 33 – Expliquer pourquoi la distribution des n cases occupées parmi les m cases disponible ne correspond pas à choisir une des $\binom{m}{n}$ configurations aléatoirement de manière uniforme.

Fin de l'épreuve