TD6: Structures de données
Corrigé

MP2I Lycée Pierre de Fermat

Exercice 1. *Notation Polonaise Inversée*

Q1. Évolution de la pile au fur et à mesure que l'on lit l'expression :

2
$2, d$
$d - 2$
$d - 2, c$
$c - (d - 2)$
$c - (d - 2), b$
$c - (d - 2), b, a$
$c - (d - 2), a + b$
$(a + b) \times (c - (d - 2)$

Q2.

(a) $a\ 2\ c\ d\ -\times +\ 4\ b - /$ est **valide**, sa version infixe serait $(b - 4)/(((d - c) \times 2) + a)$.
(b) $x + y$ n'est **pas valide** : lorsqu'on lit le $+$, la pile ne contient qu'un élément.
(c) $x\ y + z$ n'est **pas valide** : Il reste deux éléments dans la pile à la fin.
(d) $x\ y + z\times$ est **valide**, et correspond à $z \times (y + x)$.

Q3.

(a) $a \times c + b$ donne $b\ c\ a\ \times\ +$
(b) $a \times (c + b)$ donne $c\ b\ +\ a\ \times$
(c) $2 \times (a \times a \times a + 1)$ donne $1\ a\ a\ a\ \times\ \times\ +\ 2\ \times$
(d) $(1 - x) \times (2 + (x/(1 - x)) + x)$ donne $x\ x\ 1\ -\ x\ /\ +2\ +\ x\ 1\ -\ times$

Q4. Dans une expression en NPI, on a $N_{op} = N_c - 1$.
Justification : dans l'algorithme de lecture, la lecture d'une constante augmente la taille de pile de 1, et la lecture d'un opérateur diminue la taille de pile de 1. Au départ, la pile est vide, et si l'expression est valide alors la pile est de taille 1 en fin de lecture, d'où $0 + N_c - N_{op} = 1$.

Cette condition n'est pas suffisante, en effet elle serait aussi vérifiée par n'importe quelle expression infixe, comme $x + y$, qui n'est pas valide en NPI.

Q5. La lecture d'un opérateur d'arité r a pour effet de modifier la taille de pile par $1-r$. En considérant les constantes comme des opérateurs d'arité 0, on peut généraliser la formule précédente :

$$0 + \sum_{k=1}^{n} (1 - a(o_k)) = 1$$

Q6. SWAP n'a aucun effet sur la taille de la pile, et DUP se comporte comme une constante et augmente la taille de pile. On peut introduire la notion de **co-arité**, défini comme le nombre d'éléments produits par un opérateur. Par exemple, DUP et SWAP sont de co-arité 2. Alors, en notant $o_1, \ldots, o_n$ les opérateurs de l'expression, et en notant $c(o_i)$ la co-arité de l'opérateur o_i, on a :

$$0 + \sum_{i=1}^{n} (c(o_i) - a(o_i)) = 1$$

car chaque opérateur o_i consomme $a(o_i)$ et produit $c(o_i)$.

Q7. Une solution en 13 coups : 4 π 3 $\times$ / **DUP** 3 + **SWAP** sin 1 + /

Q8. Les conditions exprimées précédemment traduisent le fait que la pile doit avoir une taille de 1 à la fin de l'algorithme de lecture d'une expression en NPI. Or, pour que la lecture se passe correctement, il faut aussi que **chaque** opérateur lu puisse s'exécuter correctement. Ainsi, lorsqu'on lit l'opérateur o_k, il faut que la pile contienne au moins $a(o_k)$ éléments, c'est à dire qu'on ait :

$$\sum_{i=1}^{k-1} (c(o_i) - a(o_i)) \leq a(o_k)$$

Si chaque opérateur peut être lu, et que la pile contient bien un seul élément à la fin, l'expression est valide. Donc, l'ensemble suivant de conditions est nécessaire et suffisant :

$$\sum_{i=1}^{n} (c(o_i) - a(o_i)) = 1 \tag{1}$$

$$\forall k \in [\![1, n]\!], \sum_{i=1}^{k-1} (c(o_i) - a(o_i)) \leq a(o_k) \tag{2}$$

Exercice 2. *Opérations de pile*

Q1. L'algorithme renvoie une pile contenant les éléments de P dans l'ordre inverse, **et vide** P.

Q2. L'idée est de renverser la pile deux fois, en faisant une copie des éléments dans une nouvelle pile la deuxième fois :

Algorithme 1 : `copie`(P)

Entrée(s) : P une pile
Sortie(s) : Q copie de P, P reste inchangée

```
1 T ← Renverser(P);
2 C ← pile_vide()// Pile copie
3 tant que T n'est pas vide faire
4 |   x ← T.dépiler();
5 |   P.empiler(x) ;
6 |   C.empiler(x) ;
7 retourner C
```

Q3. L'idée est de sauvegarder P en même temps que l'on calcule la taille, afin de pouvoir restaurer les éléments à la fin :

Algorithme 2 : `taille`(P)

Entrée(s) : P une pile
Sortie(s) : taille de P

```
1 S ← pile_vide()// pile de sauvegarde
2 c ← 0;
3 tant que P n'est pas vide faire
4 |   x ← dépiler(P);
5 |   empiler(S, x) ;
6 |   c ← c + 1;
7 Renverser S dans P;
8 retourner c
```

Notons qu'un autre possibilité est de copier la pile P au début, puis de travailler sur la copie, que l'on a donc pas besoin de sauvegarder :

```
1 C ← copie(P) ; c ← 0;
2 tant que C n'est pas vide faire
3 |   x ← dépiler(C);
4 |   c ← c + 1;
5 retourner c
```

En pratique, ce deuxième algorithme serait un peu plus lent, même s'il reste en complexité $\mathcal{O}(n)$.

Q4. Une première version simple consiste à créer des copies des piles, on peut ensuite les dépiler sans s'embêter à devoir sauvegarder les valeurs :

On peut aussi mettre en place un mécanisme de sauvegarde comme dans les algorithmes précédents, ce qui donne une version un peu plus optimisée :

Les deux algorithmes sont en $\mathcal{O}(n)$ avec n la somme des tailles des piles, mais le premier s'exécute toujours en $\Theta(n)$ puisqu'il commence par copier les piles, alors que le deuxième algorithme est plus adaptatif. Par exemple, si les deux piles ont des sommets différents, le deuxième algorithme va s'exécuter en $\mathcal{O}(1)$.

Algorithme 3 : Comparer

Entrée(s) : P, Q deux piles
Sortie(s) : Booléen indiquant si $P = Q$

```
1  P' ← copie(P) ; Q' ← copie(Q) ; // Invariant: P = Q si et seulement si
       P' = Q' (en notant P_0, Q_0 les états initiaux)
2  tant que P' n'est pas vide ET Q' n'est pas vide faire
3      x ← dépiler(P');
4      y ← dépiler(Q');
5      si x ≠ y alors
6          retourner faux
   // Invariant: en sortie, P ou Q est vide, donc P = Q si et seulement
       si P' ET Q' sont vides.
7  retourner est_vide(P') ET est_vide(Q')
```

Algorithme 4 : Comparer

Entrée(s) : P, Q deux piles
Sortie(s) : Booléen indiquant si $P = Q$

```
1  S ← pile_vide()// Pile de sauvegarde
   // Invariant: P_0 = Q_0 si et seulement si P = Q (en notant P_0, Q_0 les
       états initiaux)
   // Invariant: renverser S sur P donne P_0, pareil pour Q et Q_0
2  v ← vrai // Les piles sont-elles égales
3  tant que P n'est pas vide ET Q n'est pas vide ET v faire
4      x ← dépiler(P);
5      y ← dépiler(Q);
6      si x ≠ y alors
7          v ← faux;
8          empiler(P, x);
9          empiler(Q, y);
10     sinon
11         empiler(S, x)// Remarque: x = y à ce moment
   // Invariant: en sortie, s'il n'y a pas eu de cas d'inégalité, alors P
       ou Q est vide, donc P_0 = Q_0 si et seulement si P ET Q sont vides.
12 si NON(est_vide(P)) OU NON (est_vide(Q)) alors
13     v ← faux;
14 Renverser S sur P et Q;
15 retourner v
```

Q5. Pour réaliser cette opération en $\mathcal{O}(1)$ dans l'implémentation par tableaux, il suffit de se rappeler que dans l'implémentation par tableau, un des attributs de la structure concrète est précisément la taille de la pile, il suffit alors de renvoyer la valeur de cet attribut :

```
1  int taille(pile_t* p){
2    return p->taille;
3  }
```

Q6. On parcourt la liste en comptant les maillons :

```
1  int taille(pile_t* p){
2    int l = 0;
3    for (maillon_t* m = p->sommet; m != NULL; m = m->suivant){
4      l++;
5    }
6    return l;
7  }
```

Q7. Pour implémenter l'opération taille en $\mathcal{O}(1)$ sur les listes chaînées, on pourrait **rajouter un attribut** `.taille`, que l'on met à jour à chaque opération sur la pile :

```
1  struct pile {
2    int taille;
3    maillon_t* sommet;
4  }
5  ...
6  void empiler(pile_t* p, int x){
7    p->taille++;
8    ...
9  }
10 ...
11 int depiler(pile_t* p){
12   p->taille--;
13   ...
14 }
```

Les complexités des fonctions empiler et dépiler restent $\mathcal{O}(1)$, et l'opération taille est maintenant elle aussi en $\mathcal{O}(1)$!