3. La structure de pile

I - Notion de type abstrait de données

1) Généralités

Pour spécifier un type de données, la description de la nature des valeurs stockées dans des instances de ce type ne suffit pas. Il faut également spécifier les opérations permettant de manipuler les objets de ce type (par exemple les opérations arithmétiques pour le type "entier").
Pour conserver un maximum de généralité, on donne en général pour chaque type de données un "petit" nombre d'opérations, dites *primitives*, censées permettre d'obtenir par "combinaison" toutes les opérations utiles. Ce petit nombre n'est pas toujours minimal : par exemple, pour les nombres entiers, on inclut en général la multiplication parmi les primitives (alors qu'elle peut être obtenue à l'aide de l'addition !).
L'intérêt de cette notion de *type abstrait de données* est de permettre la rédaction d'ouvrages théoriques décrivant des algorithmes utilisant ces types, indépendamment de tout langage de programmation. Les concepteurs des langages dits *de haut niveau* s'attachent à implémenter le stockage des valeurs et les primitives associées pour les types de données les plus classiques.
Comme souvent le diable est dans les détails, les choix sont rarement universels et il faut parfois s'adapter aux particularités de tel ou tel langage...

2) L'exemple du type "liste"

a) Les listes en informatique théorique

<u>*Les valeurs à stocker*</u> : E étant un ensemble non vide, l'ensemble des *listes d'éléments de* E est

$$\mathcal{L} = \bigcup_{n\in\mathbb{N}} \mathcal{E}_n \quad \text{où} \quad \mathcal{E}_0 = \{\emptyset\} \quad \text{et} \quad \forall n \geq 1 \quad \mathcal{E}_n = E \times \mathcal{E}_{n-1} \ .$$

$\mathcal{E}_n$ est l'ensemble des listes de *longueur* n.

Une liste est donc :

- soit $\emptyset$ (*liste vide*, de longueur 0) ;
- soit une liste de longueur $n \geq 1$, de la forme $\ell = (t, q)$ où t est un élément de E (la *tête* de ℓ) et q une liste de longueur $n-1$ (la *queue* de ℓ).

NB : cette définition possède un aspect récursif...

<u>*Les primitives*</u> : pour pouvoir utiliser ces listes, il suffit de disposer des fonctions suivantes :

- `est_vide(lst)` : renvoie `Vrai` si la liste `lst` est vide, `Faux` sinon ;
- `liste_vide()` : renvoie la liste vide ;
- `cons(t,q)` : renvoie la liste de tête `t`, de queue `q` ;
- `tete(lst)` : renvoie la tête de la liste **non vide** `lst` ;
- `queue(lst)` : renvoie la queue de la liste **non vide** `lst`.

Avec cette définition, on ne peut accéder directement qu'à l'élément en tête d'une liste, on accède aux éléments suivants de proche en proche ; on parle de *liste chaînée* ; une telle liste est une structure de données à *accès séquentiel*, par opposition aux structures à *accès aléatoire* qui permettent l'accès direct à un élément quelconque, comme les tableaux. Comme l'élément en tête est le dernier à avoir été ajouté à la liste (par un appel à `cons`), on parle de structure de type LIFO (*last in, first out*). Il existe aussi des structures de type FIFO (*first in, first out*), comme les files d'attente...
Historiquement, l'intérêt des listes chaînées était la gestion *dynamique* de la mémoire, la place en mémoire d'une liste évoluant selon sa longueur, tandis que les tableaux étaient gérés de façon *statique*, leur taille étant déterminée à leur création.

b) Les "listes" Python

Certains langages récents (Python, mais aussi Maple, etc.) ont pris des libertés avec cette spécification standard des listes en confondant plus ou moins les structures de liste et de tableau. Ainsi, ce que la documentation de Python appelle "*lists*" s'utilise comme des tableaux, au sens où Python permet l'accès aléatoire aux valeurs stockées, grâce à l'indexation. Toutefois le stockage est bien géré de façon dynamique, au sens où l'on peut faire varier la taille en cours d'exécution d'un programme.

3) Autres exemples

Divers types abstraits de données sont présents dans la "littérature" informatique : les *files d'attente* déjà citées, les *ensembles*, les *arbres*, les *graphes* sont hors programme mais nous allons nous intéresser aux *piles*.

II - Le type "pile" (en anglais *stack*)

1) Spécification théorique

Les valeurs à stocker : E étant un ensemble non vide, mathématiquement parlant l'ensemble des *piles d'éléments de* E est par définition égal à l'ensemble des listes d'éléments de E (cf. § **I-2)a)**), mais le vocabulaire est un peu différent.

Une pile est :

- soit $\emptyset$ (*pile vide*, de profondeur 0) ;
- soit une pile de profondeur $n \geq 1$, de la forme $p = (s, q)$ où s est un élément de E (le *sommet* de p) et q une pile de profondeur $n - 1$.

Les primitives : pour pouvoir utiliser les piles, il suffit de disposer des fonctions suivantes :

- `est_vide(p)` : renvoie `Vrai` si la pile `p` est vide, `Faux` sinon ;
- `pile_vide()` : renvoie la pile vide ;
- `empiler(s,p)` : ajoute `s` au sommet de la pile `p` (en anglais *push*) ;
- `depiler(p)` : renvoie et supprime le sommet de la pile **non vide** `p` (en anglais *pop*) ;
- `sommet(p)` : renvoie (sans le supprimer) le sommet de la pile **non vide** `p` (en anglais *peek* ou *top*).

NB : noter que *a priori* `empiler(s,p)` ne renvoie pas à proprement parler un résultat, mais modifie la pile `p`. Selon le langage on peut l'utiliser ainsi ou bien renvoyer la pile modifiée.
Noter aussi que — chez certains auteurs — `depiler` se contente de supprimer le sommet sans renvoyer sa valeur, ce qui oblige souvent à enchaîner `sommet` et `depiler`.

Une pile est ainsi — comme une liste chaînée — une structure LIFO, mais *sans parcours possible* (à moins de détruire la pile !) : on ne peut accéder qu'à l'élément situé au sommet.

Les piles sont couramment utilisées par de nombreux logiciels, souvent de manière interne, par exemple pour gérer le retour à la page précédente dans un navigateur Internet, la fonction "Édition/Annuler" dans un logiciel de bureautique et bien sûr pour la gestion de la récursivité !

2) Implémentation en Python

La structure naturelle pour stocker une pile en Python est la liste (au sens Python du terme !).
Pour optimiser l'implémentation, il est habile de stocker les valeurs de la pile "à l'envers" dans la liste Python (le sommet sera le dernier élément...). En effet, pour `empiler`, l'insertion d'un élément en tête d'une liste de longueur n a une complexité temporelle de l'ordre de n (il faut décaler les références de tous les éléments), tandis que nous disposons de la méthode `append` qui permet d'ajouter en temps constant un élément à la fin d'une liste (plus précisément, c'est en temps constant *amorti* : l'ajout de k éléments à une liste de longueur n a une complexité au pire de l'ordre de $k + n$).

L'accès au sommet est évident avec l'indexation, pour `depiler` la suppression d'un élément peut se faire à l'aide de la commande `del` ou de la méthode `pop` (qui renvoie l'élément supprimé, comme par hasard !).

```
1  def pile_vide():
2      return []

3  def est_vide(p):
4      return p==[]

5  def sommet(p):
6      return p[-1]
```

```
7  def empiler(s,p):
8      p.append(s)
9      return p

10 def depiler(p):
11     return p.pop()
```

NB : selon la remarque du § **II-1),** la commande `empiler` (*push*) **modifie** la pile transmise en paramètre, le `return p` est donc facultatif, mais permet d'éviter de mauvaises surprises à l'utilisateur...

III - Application à l'évaluation des expressions arithmétiques

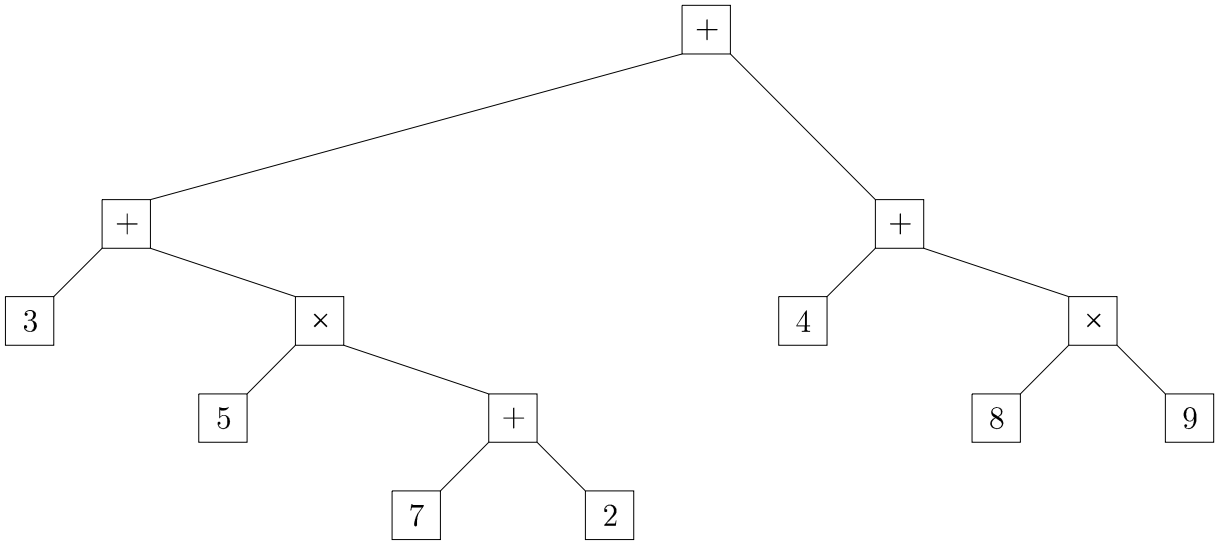
1) Représentation d'une expression arithmétique

Pour simplifier, nous nous limiterons ici à des enchaînements d'additions et de multiplications de deux nombres entiers. Considérons par exemple l'expression

$$E = (3 + (5 \times (7 + 2))) + (4 + (8 \times 9)) .$$

Cette écriture, dite *écriture infixée totalement parenthésée*, est allégée — dans l'usage courant — de quelques parenthèses, compte tenu de la convention de priorité de la multiplication sur l'addition et aussi du fait que l'on accepte les sommes de plusieurs nombres. Mais, en l'absence de priorités et si l'on se limite à des opérateurs *binaires*, les parenthèses ci-dessus sont nécessaires pour définir l'ordre des opérations.

On peut aussi représenter une telle expression par un arbre, chaque sous-arbre correspondant à une sous-expression entre parenthèses. L'expression E serait représentée par l'arbre suivant :



Une autre représentation courante est l'écriture *postfixée* (également appelée *notation polonaise inversée*, RPN en anglais, en hommage à son "inventeur" Jan Łukasiewicz, logicien et philosophe polonais). Dans cette écriture, les opérandes sont écrits l'un après l'autre, suivis de l'opérateur (l'écriture infixée $a + b$ devient $a\ b+$). Partant de la représentation par un arbre, on écrit (récursivement !) l'expression postfixée associée au sous-arbre gauche, suivie de l'expression postfixée associée au sous-arbre droit, suivie de l'opérateur à la racine de l'arbre (cela pour un opérateur *binaire*, *i.e.* à deux arguments...) ; on parle de *parcours postfixe* de l'arbre.
Pour l'arbre précédent, cela donne

$$3\quad 5\quad 7\quad 2\ +\ \times\ +\ 4\quad 8\quad 9\ \times\ +\ +$$

On montre (voir l'annexe ci-dessous) que cette écriture est non ambiguë, même sans parenthèses. Par exemple, l'expression infixée $3+4*5$ est ambiguë, mais les expressions postfixées associées à $(3+4)*5$ et à $3+(4*5)$ sont respectivement

$$3 \quad 4 \quad + \quad 5 \quad * \quad \text{et} \quad 3 \quad 4 \quad 5 \quad * \quad +$$

L'écriture postfixée est utilisée dans certains langages (Forth, Postscript,...). Symétriquement on peut définir l'écriture *préfixée*, utilisée notamment dans le langage Lisp.

2) Évaluation à l'aide d'une pile

Par sa structure même, l'écriture postfixée d'une expression permet une évaluation très simple à l'aide d'une pile (d'où son intérêt par rapport à l'écriture infixée avec parenthèses, qui nécessite une analyse syntaxique...).

L'algorithme est le suivant : partant d'une pile vide,on lit successivement les "termes" de l'expression postfixée et

- si l'on trouve un nombre on l'empile
- si l'on trouve un opérateur unaire f (par exemple le *moins unaire* ou une fonction d'une variable), on remplace le sommet x de la pile par $f(x)$
- si l'on trouve un opérateur binaire ω (par exemple une opération comme $(x,y) \mapsto x+y$), on remplace les deux valeurs x, y au sommet de la pile par $\omega(x,y)$.

Cela suppose que les identificateurs des opérateurs unaires et binaires soient distincts (attention au symbole $-$!) et l'on peut facilement généraliser à des opérateurs d'arité supérieure à 2...

La programmation explicite en Python sera effectuée en TD.

IV - Annexe hors programme

L'algorithme ci-dessus suppose que l'expression postfixée fournie est syntaxiquement correcte, ce qui ne se voit pas forcément au premier coup d'œil... L'étude ci-dessous fournit un outil de contrôle, qui permet au passage de démontrer la non ambiguïté de la notation postfixée. On en profite pour bien séparer l'aspect *syntaxe* (l'ordonnancement des symboles) de l'aspect *sémantique* (le sens donné aux symboles au moment de l'évaluation).

1) Syntaxe des expressions arithmétiques postfixées

a) Définitions - Notations

On se propose de définir les *expressions arithmétiques postfixées* (syntaxiquement correctes !).
Soient E un ensemble (de "nombres"), Ω un ensemble d'applications de $E \times E$ dans E (*opérateurs binaires*, dits aussi d'*arité* 2) et $\mathcal{F}$ un ensemble d'applications de E dans E (*opérateurs unaires*).
On note $\mathcal{S} = E \cup \Omega \cup \mathcal{F}$ l'ensemble des *"symboles"* utilisés et $\mathcal{L}$ l'ensemble des listes non vides d'éléments de $\mathcal{S}$. Pour alléger, on écrit les éléments d'une telle liste simplement séparés par des espaces.
Si $A = a_1 \ \dots \ a_p$ et $B = b_1 \ \dots \ b_q$ sont deux éléments de $\mathcal{L}$, on note $A\,B$ le résultat de la *concaténation* des deux listes, à savoir

$$A\,B = a_1 \ \dots \ a_p \ b_1 \ \dots \ b_q \ .$$

L'ensemble $\mathcal{E}$ des expressions arithmétiques postfixées est le sous-ensemble de $\mathcal{L}$ défini récursivement comme l'ensemble des éléments de $\mathcal{L}$ s'écrivant sous l'une des formes suivantes :

- x *(liste de longueur* 1*)*, où $x \in E$;
- $A\,B\,\omega$, où $(A,B) \in \mathcal{E}^2$ et $\omega \in \Omega$;
- $A\,f$, où $A \in \mathcal{E}$ et $f \in \mathcal{F}$.

NB : on peut bien sûr généraliser cette construction à des expressions comprenant des opérateurs d'arité supérieure à 2.

Définition : étant donné $A = a_1 \ \ldots \ a_n$ élément de $\mathcal{L}$, on appelle :

* *poids* de A l'entier $p(A) = \sum_{k=1}^{n} p(a_k)$ où l'on a posé, pour $a \in \mathcal{S}$, $p(a) = \begin{cases} 1 & \text{si } a \in E \\ -1 & \text{si } a \in \Omega \\ 0 & \text{si } a \in \mathcal{F} \end{cases}$.
* *préfixes stricts* de A les listes de la forme $a_1 \ \ldots \ a_p$, $1 \leq p < n$.

b) Caractérisation des expressions arithmétiques postfixées

Théorème : un élément A de $\mathcal{L}$ est dans $\mathcal{E}$ si et seulement si A est de poids 1 et tout préfixe strict de A est de poids supérieur ou égal à 1.
Si $A = a_1 \ \ldots \ a_n$ est dans $\mathcal{E}$, avec $n > 1$, alors A s'écrit, de manière unique,

* soit $A = B\ C\ \omega$, avec $(B, C) \in \mathcal{E}^2$ et $\omega \in \Omega$,
* soit $A = B\ f$, avec $B \in \mathcal{E}$ et $f \in \mathcal{F}$.

Dans les deux cas, B est le plus long préfixe strict de poids 1 de A.

Démonstration

1) Condition nécessaire : je montre par récurrence sur n que la propriété $\mathcal{P}_n$: "si A est un élément de $\mathcal{E}$ de longueur au plus égale à n, alors A est de poids 1 et tout préfixe strict de A est de poids supérieur ou égal à 1" est vraie pour tout n de $\mathbb{N}^*$.

a) $\mathcal{P}_1$ est claire : un élément de $\mathcal{E}$ de longueur 1 est nécessairement de la forme x, $x \in E$.

b) soit $n > 1$ tel que $\mathcal{P}_{n-1}$ soit vraie ; il suffit de prouver le résultat pour $A \in \mathcal{E}$, de longueur n :

* si $A = B\ C\ \omega$, avec $(B, C) \in \mathcal{E}^2$ et $\omega \in \Omega$, alors : $p(A) = p(B) + p(C) - 1 = 1 + 1 - 1 = 1$
  cela grâce à l'hypothèse de récurrence appliquée à B et à C.
  De plus, les préfixes stricts de A sont B, $B\ C$, et les listes, soit de la forme B', avec B' préfixe strict de B, soit de la forme $B\ C'$, avec C' préfixe strict de C : dans ces quatre cas l'hypothèse de récurrence permet de conclure ;
* si $A = B\ f$, avec $B \in \mathcal{E}$ et $f \in \mathcal{F}$, alors : $p(A) = p(B) + 0 = 1$,
  cela grâce à l'hypothèse de récurrence appliquée à B.
  De plus, les préfixes stricts de A sont B et les préfixes stricts de B, d'où le résultat grâce à l'hypothèse de récurrence.

2) Soit A élément de $\mathcal{E}$ de longueur strictement supérieure à 1 :

a) si $A = B\ f$, où $B \in \mathcal{E}$, $f \in \mathcal{F}$, alors B est clairement le plus long préfixe strict de poids 1 de A

b) si $A = B\ C\ \omega$, avec $(B, C) \in \mathcal{E}^2$ et $\omega \in \Omega$, alors B est le plus long préfixe strict de poids 1 de A (en effet B est bien un préfixe strict de poids 1 de A et tout préfixe strict de A plus long que B est de poids supérieur ou égal à 2 d'après la condition nécessaire établie ci-dessus).

Il en résulte que l'écriture d'un élément A de $\mathcal{E}$ de longueur strictement supérieure à 1 sous l'une des deux formes $B\ f$ ou $B\ C\ \omega$ est unique puisque B est parfaitement défini, dans les deux cas, par la propriété précédente ; C s'en déduit immédiatement dans le second cas (cette constatation correspond à la non-ambiguïté de l'écriture postfixée, contrairement à l'écriture infixée qui nécessite des parenthèses).

3) Condition suffisante : je montre par récurrence sur n que la propriété $\mathcal{P}_n$: "si A est un élément de $\mathcal{L}$, de longueur au plus égale à n, tel que A est de poids 1 et tout préfixe strict de A est de poids supérieur ou égal à 1, alors A appartient à $\mathcal{E}$" est vraie pour tout n de $\mathbb{N}^*$.

a) $\mathcal{P}_1$ est vraie : si A est un élément de $\mathcal{L}$ de longueur 1 et de poids 1, nécessairement, d'après la définition de p, A est de la forme x, $x \in E$ et donc $A \in \mathcal{E}$;

b) soit $n > 1$ tel que $\mathcal{P}_{n-1}$ soit vraie ; soit $A = a_1 \ \ldots \ a_n$ élément de longueur n de $\mathcal{L}$, de poids 1 et dont les préfixes stricts sont de poids supérieur ou égal à 1. En particulier $p\,(a_1 \ \ldots \ a_{n-1}) \geq 1$ et donc $p\,(a_n) \leq 0$, c'est-à-dire que a_n est un opérateur, d'où les deux cas :

- si $a_n = f \in \mathcal{F}$, alors $A' = a_1 \ \ldots \ a_{n-1}$ est de poids 1 et ses préfixes stricts sont de poids supérieur ou égal à 1 (puisque ce sont des préfixes stricts de A !) ; par conséquent $A' \in \mathcal{E}$ d'après $\mathcal{P}_{n-1}$ et donc $A = A'\ f \in \mathcal{E}$;
- si $a_n = \omega \in \Omega$, alors $A' = a_1 \ \ldots \ a_{n-1}$ est de poids 2 ; a_1 est nécessairement de poids 1 (supérieur ou égal à 1 car a_1 est un préfixe strict de A, inférieur ou égal à 1 par définition de p) et donc A admet un préfixe strict B de poids 1 et de longueur maximale (le plus grand élément de la partie non vide majorée de $\mathbb{N}$ formée des longueurs des préfixes stricts de A de poids 1) ; B est élément de $\mathcal{E}$ grâce à l'hypothèse de récurrence (ses préfixes stricts sont des préfixes stricts de A), en outre B est un préfixe strict de A' puisque $p\,(A') = 2$; A' est donc de la forme $B\ C$, où C est un élément de $\mathcal{L}$ de poids $p\,(A') - p\,(B)$, donc de poids 1 ; enfin, si C' est un préfixe strict de C, alors $B\ C'$ est un préfixe strict de A strictement plus long que B, donc de poids supérieur ou égal à 2 par définition de B, d'où
$$p\left(C'\right) = p\left(B\ C'\right) - p\,(B) \geq 2 - 1 = 1\ ;$$
par conséquent l'hypothèse de récurrence s'applique également à C, donc C et par suite $A = B\ C\ \omega$ sont des éléments de $\mathcal{E}$.

Cela achève la démonstration du théorème.

2) Sémantique

On se propose maintenant d'*évaluer* les expressions arithmétiques postfixées (il s'agit de donner un sens à des objets syntaxiquement corrects).

Théorème : il existe une unique application eval de $\mathcal{E}$ dans E telle que :

- $\mathrm{eval}\,(A) = x$ si $A = x$, où $x \in E$;
- $\mathrm{eval}\,(A) = \omega\left(\mathrm{eval}\,(B)\,, \mathrm{eval}\,(C)\right)$ si $A = B\ C\ \omega$, où $(B, C) \in \mathcal{E}^2$ et $\omega \in \Omega$;
- $\mathrm{eval}\,(A) = f\left(\mathrm{eval}\,(B)\right)$ si $A = B\ f$, où $B \in \mathcal{E}$ et $f \in \mathcal{F}$.

Démonstration
Le résultat se prouve par récurrence sur la longueur de A, en utilisant le fait (vu au paragraphe précédent) que A s'écrit de manière unique sous l'une des trois formes x, $B\ C\ \omega$, $B\ f$.