Piles

1 Pile d'exécution d'un programme récursif

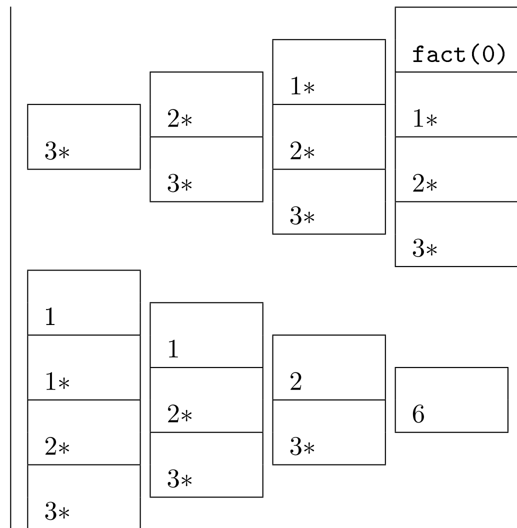
Reprenons le calcul récursif de la factorielle.

```
def fact(n):
    if n == 0:
        return 1
    return n * fact(n-1)
```

Voyons comment s'effectue le calcul de 3!

Appel à `fact(3)`
- `3*fact(2)` = ?
- Appel à `fact(2)`
  - `1*fact(1)`= ?
  - Appel à `fact(1)`
    - `1*fact(0)` = ?
    - Appel à `fact(0)`
    - Retour de la valeur 1
    - 1*1
  - Retour de la valeur 1
  - 2*1
- Retour de la valeur 2
- 3*2

Retour de la valeur 6.



La pile d'exécution de la fonction est un emplacement en mémoire destiné à stocker les paramètres, les variables locales et les adresses de retour des fonctions en cours d'exécution. Cette pile fonctionne sur le principe *LIFO (Last In First Out)*. Dernier entré, premier sorti.

2 Complément sur la récursivité : récursivité terminale

Notons que cette double exécution, ascendante puis descendante, peut être contournée. Voici une autre version du calcul récursif de la factorielle.

```
def fact_term(n, a):
    if n <= 1:
        return a
    return fact_term(n-1,n*a)
```

```
fact_term(3,1)
    fact_term(2,3)
        fact_term(1,6)
            6.
```

Une fonction récursive `fonction` dont les appels sont de la forme `fonction(·)`, sans qu'il n'y ait donc d'opération sur cette valeur, est dite *récursive terminale.* Il n'y a donc rien à garder en mémoire dans la pile d'exécution. Une fonction récursive terminale est en théorie plus efficace (mais souvent plus difficile à écrire) qu'une fonction récursive non terminale : il n'y a qu'une phase de descente et pas de phase de remontée. Les appels n'ont pas besoin d'être empilés et chaque appel remplace simplement l'appel précédent.

Il est facile de dérécursifier une fonction récursive terminale en la remplaçant par une fonction itérative. Dans le cas général, c'est possible mais parfois très compliqué ; le problème des tours de Hanoï, notamment, est délicat à dérécursifier.

```
def recursif(x):
    instructions I0
    if condition C0 :
        instructions I1
    else:
        instructions I2
        recursif(f(x))
```

```
def iteratif(x):
    instructions I0
    while not C0:
        instructions I2
        x = f(x)
        instructions I0
    instructions I1
```

3 Expression bien parenthésée

Nous étudions dans ce paragraphe un exemple de situation où un contrôle de validité est organisé sous la forme d'une pile.

Pour voir si une expression est bien parenthésée, on la parcourt de gauche à droite. À chaque parenthèse, crochet ou accolade fermante rencontrée, on vérifie que celle-ci est bien la dernière à avoir été ouverte. En fait, la validité du parenthésage se réduit à l'agencement des parenthèses, sans s'occuper des autres signes. Une expression peut donc être bien parenthésée, mais algébriquement non valide, à cause d'une division par 0, par exemple, ou d'un placement fautif des signes algébriques - c'est le cas de la dernière des trois expressions ci-dessous.

Exemples d'expression correctement parenthésées :
- $((3 + 2 * (4 - 5/6 + 4) - 1) * 5 - 1)$,
- $(2 + 3 + (4 + 5)/(6 + 7))$,
- $(2 - 5(*3 + A)) * 2 + (1/3)$.

Exemples d'expressions incorrectement parenthésées :
- $(t + u)) * 3)$,
- $((())$.

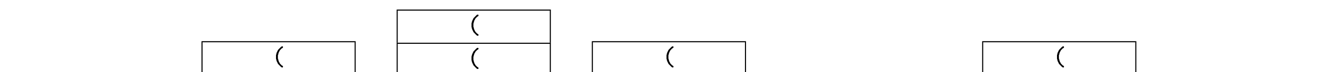
Pour voir si une expression est bien parenthésée, il suffit de vérifier :
- qu'on ne ferme pas une parenthèse avant de l'avoir ouverte ;
- qu'à a bien refermé toutes les parenthèses ouvertes.

Pour ce faire, on lit l'expression algébrique, typée comme chaîne de caractères, dans l'ordre des indices croissants (ce qui correspond à une lecture de gauche à droite, donc). Après avoir créé une pile vide, on procède comme suit : à chaque fois que l'on rencontre une parenthèse ouvrante, on la met sur le dessus de la pile. Quand on lit une parenthèse fermante, on regarde si la pile contient sur le dessus une parenthèse ouvrante (donc si la pile est non vide. Si c'est le cas, on la supprime. Dans le cas contraire, on renvoie `False` et l'on sort de la fonction. Quand on atteint la fin de la chaîne, on renvoie `True` si la pile est vide (toutes les parenthèses ouvertes ont bien été fermées) et `False` dans le cas contraire.

En représentant provisoirement la pile comme une liste, on peut écrire le programme suivant :

```
def parenthese(C):
    L = []
    for c in C:
        if L = [] and c = ')':
            return False
        elif c = '(':
            L.append(c)
        elif c = ')':
            L.pop()
    return L == []
```

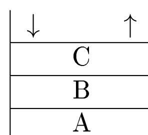
Pour la troisième expression bien parenthésée, les états successifs de la pile seront :



4 Notion de pile

La structure de pile correspond à l'image traditionnelle d'une pile de cartes ou d'assiettes posée sur une table.

En particulier, on ne peut accéder qu'au dernier élément ajouté, qu'on appelle le sommet de la pile. Ainsi, si on a ajouté successivement A, puis B, puis C dans une pile, on se retrouve dans la situation suivante :



où C est empilé sur B, lui-même empilé sur A. On peut soit retirer C de la pile (on dit qu'on dépile C), soit ajouter un quatrième élément D (on dit qu'on empile D).

Si on veut accéder à l'élément A, il faut commencer par dépiler C puis B. L'image associée à une pile est donc *dernier arrivé, premier sorti* (en anglais *Last In, First Out* ou *LIFO*).

5 Fonctions sur les piles

Compte tenu de la structure que l'on souhaite mettre en place, nous allons définir des fonctions sur les piles. Il existe quelques variantes de cette implémentation, que nous présenterons ensuite :

`creer_pile()` : cette fonction va créer une pile vide.

`test_vide(p)` : cette fonction va tester si la pile p est vide et renvoyer un booléen.

`pop(p)` : la fonction **pop** désempile le sommet de la pile et renvoie sa valeur.

`push(p,x)` : la fonction **push** empile la valeur x sur le sommet de la pile. Elle ne renvoie rien.

L'ensemble de ces fonctions sont en temps constant, c'est à dire en $\mathcal{O}(1)$.

Lors des exercices sur les piles, il est **formellement interdit** d'utiliser toute autre fonction ayant trait aux manipulations de listes ou de tableaux que les fonctions ci-dessus.

Certaines implémentations des piles proposent alternativement que la fonction `pop` désempile la pile sans rien renvoyer et qu'une autre fonction sommet (en anglais `peek` permette d'accéder au sommet de la pile, sans le désempiler. Ces deux implémentations sont équivalentes.

6 Applications des piles

Nous étudierons en exercice quelques applications des piles :
- Les appels successifs à une fonction récursive sont gérés par une pile d'appel.
- La structure de pile s'adapte particulièrement bien à l'étude syntaxique de certaines expressions. On l'a déjà évoqué avec la question des expressions bien parenthésées.
- Les navigateurs web, traitements de texte, et autre logiciel disposent en général d'une fonction « défaire »ou « revenir en arrière ». De telles fonctionnalités sont traitées informatiquement par des piles.

De manière plus théorique, dans les langages de programmation compilés, comme le C, les piles servent à gérer les paramètres d'appels des fonctions, les variables locales et les points de retour.

7 Pour aller plus loin : notion de file

Dans certains cas, une structure dont le fonctionnement est proche des piles, mais où le dernier arrivé est le dernier servi est adaptée (*Last In Last Out* ou *LILO*). Il serait en effet absurde de traiter des dossiers administratifs selon le principe d'une pile même s'il se dit que certains enseignants traitent ainsi parfois leurs paquets de copies.

8 Exercices

On supposera dans les exercices que l'on dispose d'une classe pile et des méthodes suivantes :
- `P=pile()` : Créé une pile P vide.
- `P.est_vide()` : Renvoie un booléen indiquant si la pile P considérée est vide ou non ;
- `P.pop()` : Dépile le sommet de la pile P et renvoie sa valeur (ou une erreur si la pile est vide) ;
- `P.push(x)` : Empile l'élément x au sommet de la pile P.

Exercice 1. 1.Écrire une fonction qui intervertit les deux éléments situés au sommet d'une pile de taille au moins égale à 2.
2. Écrire une fonction qui permet de lire (sans l'extraire) le n-ième élément d'une pile. On prévoira le cas où la pile n'est pas de taille suffisante pour qu'un tel élément existe.
3. Écrire une fonction qui inverse une pile.
4. Écrire une fonction qui copie une pile.
5. Écrire une fonction taille, qui calcule la taille d'une pile. Quelle est sa complexité en temps, et quelle est sa complexité en mémoire ?
6. Écrire une fonction swap, qui échange le sommet et le fond d'une pile de taille au moins égale à 2.

Exercice 2. Un navigateur web doit proposer les fonctions suivantes :
- Visiter une nouvelle page web
- Revenir en arrière d'une page web
- Aller d'une page en avant

À l'aide de deux piles, écrire un ensemble de fonctions qui réponde à ce cahier des charges.

Exercice 3. **1.** Réécrire la fonction de vérification du parenthésage en utilisant les commandes spécifiques aux piles.
2. Généraliser à une expression algébrique comprenant des parenthèses, des crochets et des accolades.

Exercice 4. Proposer un ensemble de fonctions adaptées aux files. Quel est le comportement attendu de chaque fonction ?