Objectifs

- connaître les types de données récursifs *liste* et *arbre*
- savoir en donner une implantation
- savoir l'utiliser

La récursivité ne s'applique pas uniquement aux algorithmes. Dans ce cours, nous allons définir deux nouveaux types de données :

- les listes récursives ;
- les arbres.

Définition

Un type de données est dit récursif lorsque parmi les éléments qui permettent de le définir, on trouve le type de données lui-même.

1 les listes récursives

Nous traitons ici des listes récursives et non des listes prédéfinies Python qui sont une autre structure de données.

Nous savons tous intuitivement ce qu'est une liste :

1. Une liste est une collection finie d'emplacements qui se suivent et se précèdent. C'est donc une structure de données *séquentielle* ou *linéaire*.
2. Une liste peut contenir un nombre quelconque d'éléments, y compris aucun (la liste vide).
3. Dans une liste, les emplacements ne sont jamais vides.

Nous allons essayer de construire une définition plus formelle de ce qu'est une liste afin d'en dégager les opérations primitives et une structure essentielle qui nous permettra d'en donner des implantations.

Prenons une liste comme par exemple $\ell_1 = [3, 1, 4]$. C'est une liste à trois éléments (ou de longueur trois) dont le premier est 3, le deuxième 1, et le dernier 4.

Une autre façon de décrire cette liste consiste à dire que

- la liste ℓ_1 possède un premier élément 3 qu'on nommera élément de *tête*,
- et que vient après cet élément de tête la liste $\ell_2 = [1, 4]$ des éléments qui suivent, liste qu'on nommera *reste*.

Ce qu'on vient de dire de la liste ℓ_1 peut être répété pour la liste ℓ_2 qui est donc constituée :

- d'un élément de *tête* : 1,
- et d'un *reste* : $\ell_3 = [4]$.

À nouveau on peut répéter le même discours pour la liste ℓ_3 qui est donc constituée :

- d'un élément de *tête* : 4,
- et d'un *reste* : $\ell_4 = []$.

La liste ℓ_4 étant vide, elle ne possède pas d'élément de tête, et ne peut donc pas être décomposée comme nous venons de le faire à trois reprises.

Si on convient d'utiliser la notation (x, ℓ) pour désigner le couple constitué de l'élément x de tête, et du reste ℓ d'une liste, on peut alors écrire :

$$\ell_1 = (3, (1, (4, [])))$$

On conçoit aisément que ce qui vient d'être fait pour notre exemple de liste ℓ_1 peut être reproduit pour n'importe quelle liste.

On peut conclure cette approche en donnant une définition abstraite et formelle des listes d'éléments appartenant tous à un ensemble E.

Définition

Une *liste récursive* d'éléments d'un type A est
- soit la liste vide
- soit un couple (x, ℓ) constitué d'un élément x de type A et d'une liste ℓ d'éléments de type A.

1.1 Opérations primitives sur les listes récursives (non mutables)

En nous appuyant sur la définition formelle qui vient d'être établie, nous sommes en mesure de dégager les opérations primitives qui suivent.

1.1.1 Constructeur

D'après la définition, une liste est

- soit la liste vide,
- soit un couple constitué de l'élément de tête suivi de la liste des éléments qui suivent.

Le constructeur de liste doit donc permettre de produire soit une liste vide et pour cela aucun paramètre n'est nécessaire, soit une liste à partir de deux paramètres.

```
def __init__(self, *args):
    """
        build a new empty list if args is empty,
        or a list whose head is first element of args,
        and tail list is second element of args.

        précondition: len(args) in {0, 2}
             and if len(args) == 2, args[1] must be a ApLst

        raise: `ApLstError` if précondition is not satisfied
        """
```

1.1.2 Sélecteurs

Les listes non vides possèdent une tête et un reste. Il nous faut les méthodes, appelées sélecteurs, pour accéder à ces deux composantes.

```
def head(self):
    """
        return: head element of self
        raise: `ApLstError` if self is empty
        """

def tail(self) -> "ApLst":
    """
        return: tail list of self
        raise: `ApLstError` if self is empty
        """
```

1.1.3 Prédicat

Un prédicat testant la vacuité d'une liste (c'est-à-dire qui teste si elle est vide) peut s'avérer nécessaire.

```
def is_empty(self) -> bool:
    """
        return
           - True if self is empty
           - False otherwise
        précondition: none
        """
```

Remarque

Voici quelques relations qu'on peut établir à partir de ces opérations primitives.
- pour toute liste `l` et tout élément `x`, on a le reste de (x, ℓ) est ℓ et sa tête est x,
- et pour toute liste non vide, on a $(tetedel\ell, restedel\ell)$ est égal à ℓ.

1.2 Opérations primitives spécifiques aux listes mutables

Toutes les opérations primitives décrites ci-dessus permettent de travailler avec des listes non mutables, c'est-à-dire des listes dans lesquelles il est impossible

- de changer la valeur d'un élément d'une liste,
- et d'en changer la structure, en particulier sa longueur.

Les listes non mutables sont fréquentes dans les langages fonctionnels comme Haskell et OCaml.

Mais dans d'autres langages, les listes sont mutables. Par exemple en Python, on peut

- changer la valeur d'un élément d'une liste : `l[i] = x`
- changer l'ordre de tous les éléments : `l.sort()`
- augmenter la longueur d'une liste : `l.append(x)`
- diminuer la longueur d'une liste : `l.pop()` ...

Si on veut des listes mutables, il nous faut des opérations primitives supplémentaires.

1.3 Implantation des listes

Pour trouver une implantation des listes, il suffit de s'en tenir à la définition récursive d'une liste : une liste est soit la liste vide, soit un couple constitué de la tête de la liste et du reste de la liste.

Il semble assez naturel qu'un objet de classe `ApLst` soit représenté par un attribut dont la valeur est

- un tuple vide pour la liste vide,
- un couple pour une liste non vide.

1.3.1 Constructeurs

Nous choisissons un attribut `content` dont la valeur est `()` (tuple vide) pour la liste vide, ou un couple (tuple à deux éléments) pour les listes non vides.

```
def __init__(self, *args):
    """
    build a new empty list if args is empty,
    or a list whose head is first element of args,
    and tail list is second element of args.

    précondition: len(args) in {0, 2}
         and if len(args) == 2, args[1] must be a ApLst

    raise: `ApLstError` if précondition is not satisfied
    """
    if len(args) == 0:
        self.content = ()
    elif len(args) == 2:
        if isinstance(args[1], ApLst):
            self.content = (args[0], args[1])
        else:
            raise ApLstError('bad type for second argument')
    else:
        raise ApLstError('bad number of arguments')
```

1.3.2 Prédicat

Le prédicat de test de vacuité d'une liste peut s'écrire très simplement même sans savoir comment une liste vide est implantée.

```
    def is_empty(self) -> bool:
        """
        return
           - True if self is empty
           - False otherwise
        précondition: none
        """
        return len(self.content) == 0
```

1.3.3 Sélecteurs

Les sélecteurs sont simples à écrire, mais il faut tenir compte du cas des listes vides par le biais d'un traitement de l'exception par exemple.

```
    def head(self):
        """
        return: head element of self
        raise: `ApLstError` if self is empty
        """
        if self.is_empty():
            raise ApLstError('head: empty list')
        else:
            return self.content[0]

    def tail(self) -> "ApLst":
        """
        return: tail list of self
        raise: `ApLstError` if self is empty
        """
        if self.is_empty():
            raise ApLstError('head: empty list')
        else:
            return self.content[1]
```

1.3.4 Exemples d'utilisation de ces opérations

Voici quelques instructions de manipulation de listes :

```
>>> from aplst import ApLst, ApLstError
>>> l = ApLst(3, ApLst())
>>> l.head()
3
>>> l.is_empty()
False
>>> l.tail().is_empty()
True
```

1.3.4.1 Calcul de la longueur d'une liste

Voici une implantation possible d'une méthode donnant la longueur d'une liste.

```
def __len__(self) -> int:
    if self.is_empty():
        res = 0
    else:
        res = 1 + len(self.tail())
    return res
```

Remarque

Le calcul de la longueur effectué par la fonction ci-dessus nécessite un parcours complet de la liste. On pourrait envisager une implantation des listes avec un attribut contenant la longueur de la liste qui permettrait à la méthode `__len__` de ne pas avoir à parcourir l'intégralité de la liste.

1.3.4.2 Représentation sous forme d'une chaîne de caractères

```
    def __str__(self) -> str:
        """
        return: a string representation of list self
        précondition: none
        """
        def str_content(self, item_number=0):
            if self.is_empty():
                return ''
            elif item_number == 50:
                return ', ...'
            else:
                comma = '' if item_number == 0 else ', '
                return (comma + str(self.head()) +
                        str_content(self.tail(), item_number + 1))

        return f'[{str_content(self)}]'
```

```
>>> str(ApLst())
'[]'
>>> print(ApLst(1, ApLst(2, ApLst(3, ApLst()))))
[1, 2, 3]
```

2 Les arbres, en informatique

Un arbre est une structure de données récursive non linéaire (contrairement aux listes) utilisée pour représenter des données structurées hiérarchiquement, par exemple :

- un système de fichiers,
- un site web (arbre DOM -*Document Object Model*-),
- un arbre généalogique,
- une taxonomie.

Un *arbre* est un ensemble de nœuds connectés par des arêtes. Un *nœud* est caractérisé par :

- une valeur (on parle d'*étiquette*),
- un nombre finis d'enfants, qui peut être nul.

Une *arête* relie deux nœuds. Chaque nœud, à l'exception de la racine, est relié à un autre nœud, son parent, par exactement une arête entrante. Chaque nœud peut avoir aucune, une ou plusieurs arêtes sortantes le reliant à ses enfants.

2.1 Vocabulaire

- la *racine* d'un arbre est le seul nœud sans parent,
- les *enfants* (ou descendants) sont l'ensemble des nœuds reliés à un même nœud par des arêtes entrantes,
- le *parent* est le nœud relié à ses nœuds enfants par une arête sortante,
- une *feuille* est un nœud sans enfant,
- un nœud interne est un nœud qui n'est pas une feuille,
- un *sous-arbre* est l'ensemble des nœuds et arêtes d'un nœud parent et de ses enfants.
- un *chemin* est une liste de nœuds reliés par des arêtes.

Pour assurer la cohérence de ces définitions, on considère que l'arbre vide n'est pas un nœud.

2.2 Arbre binaire

Par la suite, nous allons nous restreindre aux arbres binaires pour lesquels chaque nœud a, au plus, deux enfants appelés par convention gauche et droit.

Un arbre binaire est donc :

- soit un arbre vide,
- soit un triplet `(e, g, d)`, appelée nœud, dans lequel
  - `e` est l'étiquette de la racine de l'arbre,

- `g` est le sous-arbre gauche de l'arbre, et
- `d` est le sous-arbre droit de l'arbre.

Les sous-arbres gauche et droit d'un arbre binaire non vide sont eux-mêmes des arbres binaires. La structure d'arbre binaire est donc une structure récursive.

2.2.1 Opérations primitives sur les arbres binaires (non mutables)

2.2.1.1 Constructeur D'après la définition, un arbre binaire est soit :

- l'arbre vide,
- un triplet constitué d'une étiquette et des sous-arbres gauche et droit.

Le constructeur d'arbre binaire doit donc permettre de produire soit un arbre vide et pour cela aucun paramètre n'est nécessaire, soit un arbre binaire à partir de trois paramètres.

2.2.1.2 Sélecteurs Les arbres binaires non vides possèdent une étiquette, un sous-arbre gauche et un sous-arbre droit. Il nous faut des sélecteurs pour accéder à ces trois composantes, nous les appellerons, respectivement, `etiquette()`, `gauche()` et `droit()`.

2.2.1.3 Prédicat Un prédicat testant la vacuité d'un arbre (c'est-à-dire qui teste si il est vide) peut s'avérer nécessaire. Nous l'appellerons `est_vide()`.

2.3 Quelques mesures sur les arbres

- la *taille* d'un arbre est son nombre de nœuds ;
- la *profondeur* d'un nœud est le nombre d'arêtes entre la racine et ce nœud, donc la profondeur de la racine est nulle ;
- la *hauteur* d'un arbre est la profondeur maximale de l'ensemble des nœuds de l'arbre ;
- l'*arité d'un nœud* est le nombre d'enfants du nœud, donc elle est au maximum de 2 dans le cas d'un arbre binaire ;
- l'*arité d'un arbre* est le nombre maximal d'enfants des nœuds de l'arbre, donc elle est de 2 dans le cas d'un arbre binaire ;

Nous allons voir comment programmer deux de ces mesures.

2.3.1 La taille

```
    def taille(self) -> int:
        """Renvoie le nombre de noeuds de l'arbre binaire.

        Précondition : aucune.

        Exemple(s) :
        $$$ vide = ArbreBinaire()
        $$$ fd = ArbreBinaire("non symétrique")
        $$$ fg = ArbreBinaire("arbre")
        $$$ nid = ArbreBinaire("binaire", vide, fd)
        $$$ ab = ArbreBinaire("un", fg, nid)
        $$$ vide.taille()
        0
        $$$ fd.taille()
        1
        $$$ nid.taille()
        2
        $$$ ab.taille()
        4
        """
        if self.est_vide():
            return 0
        else:
            return 1 + self.gauche().taille() + self.droit().taille()
```

2.3.2 La hauteur

```
    def hauteur(self) -> int:
        """Renvoie la profondeur maximale de l'ensemble des noeuds de l'arbre.

        Précondition : aucune.

        Exemple(s) :
        $$$ vide = ArbreBinaire()
        $$$ fd = ArbreBinaire("non symétrique")
        $$$ fg = ArbreBinaire("arbre")
        $$$ nid = ArbreBinaire("binaire", vide, fd)
        $$$ ab = ArbreBinaire("un", fg, nid)
        $$$ vide.hauteur()
        -1
        $$$ fd.hauteur()
        0
        $$$ nid.hauteur()
        1
        $$$ ab.hauteur()
        2
        """
        if self.est_vide():
            return -1
        else:
            return 1 + max(self.droit().hauteur(), self.gauche().hauteur())
```