Séance 4 : Parseur

L1 – Université Côte d'Azur

L'objectif du sujet est d'évaluer des expressions mathématiques exprimées sous forme de chaîne. On ne s'autorisera, pour commencer, dans la formule que des valeurs entières (le résultat lui pourra être flottant) les 5 opérations `+`, `-`, `*`, `/` et `^` (puissance). Par exemple : `"1 + (2*3)"` ou même `"5*(-1+ (3*4))^7-8/5"`.

Dans un second temps, on ajoutera la gestion des variables et les opérations de comparation, `<`, `>` et `=` ainsi que les opérations logique `&`, `|` et `~`.

On fait l'hypothèse que chaque opération ne tient que sur un symbole et on s'interdira donc les opérations `<=` ou `!=`. Ce sera un bon exercice à faire à la maison de modifier le code pour permettre ces opérations.

Exercice 1 — Mise en route

Dans cette exercice, on pourra utiliser des *slices*, c'est à dire la syntaxe « `chaîne[début:fin:pas]` ».

1. Écrire une fonction `intérieur(ch)` qui renvoie la chaîne `ch` sans le premier ni dernier caractère.

```
>>> intérieur("(2+3)")
'2+3'
```

2. Écrire une fonction `suivant(ch)` qui renvoie la chaîne `ch` sans le premier caractère.

```
>>> suivant("-(2+3)")
'(2+3)'
```

3. Écrire une fonction `découpage(ch, i)` renvoyant un triplet de sous-chaînes : l'une contenant le début de `ch` jusqu'à l'indice `i` (exclu), le caractère d'indice `i` et la fin de la chaîne (à partir de `i` exclu).

```
>>> découpage("(2+3)*(4+5)", 5)
('(2+3)', '*', '(4+5)')
```

Exercice 2 — Pré-traitement

1. Écrire une fonction `suppression_espaces(ch)` qui renvoie la chaîne `ch` sans les espaces.

```
>>> suppression_espaces("( 2+ 3) * ( 4 + 7) ")
'(2+3)*(4+7)'
```

2. En mathématique, le symbole `'-'` peut avoir deux sens différents : la soustraction (comme dans `"3-2"`) ou la négation (comme dans `"3 + -2"`). L'objectif de cette question est de remplacer toutes les négations par le symbole `"_"` ; c'est le symbole souligné (ou *underscore*) situé sur la touche [8].
   Écrire une fonction `remplacement_négatif(ch)` qui renvoie une nouvelle chaîne construite à partir de `ch` en appliquant les deux règles suivantes : 1) *« Si le premier caractère est un `"-"`, on le remplace par `"_"` ; sinon, on le laisse intacte »* et 2) *« Si un caractère `"-"` suit un des caractères de la chaîne `"(+-*/^<=>&|~"`, alors on le remplace par `"_"` ; sinon, on le laisse intacte »*. On supposera que la chaîne `ch` ne contient aucun espace.

```
>>> remplacement_négatif("1+-3")
'1+_3'
>>> remplacement_négatif("1-3")
'1-3'
```

```
>>> remplacement_négatif("-1-3")
'_1-3'
>>> remplacement_négatif("-(1+2)--5")
'_(1+2)-_5'
```

Exercice 3 — Création du dictionnaire

Le problème de l'écriture usuelle est la gestion des priorités. L'expression `"1+2*3"` s'interprète comme `"1+(2*3)"` quand `"1*2+3"` doit s'interpréter comme `"(1*2)+3"`. On souhaite ajouter des parenthèses supplémentaires afin de ne plus avoir à tenir compte des priorités implicites. L'objectif est de remplacer tous les `'+'` par `')))+((('`, tous les `'*'` par `'))*(('`, tous les `'^'` par `')^('` : plus une opération est prioritaire, moins on ajoute de parenthèses. L'expression obtenue sera alors correctement parenthésée et permettra de ne plus avoir à gérer les règles de priorité. Dans cet exercice, nous allons créer le dictionnaire indiquant les règles de substitution.

1. Créer le dictionnaire contenant les opérations `"+"`, `"-"`, `"*"`, `"/"` et `"^"`.
2. Ajoutez les clés `"("` et `")"`. Il faut que le nombre de parenthèses associé à `"("` soit égal au plus grand nombre de parenthèses (ici 3 pour l'opération `"+"`) auquel on ajoute 1. Ainsi au aura `"("` qui sera associé à `"(((("`. Même raisonnement pour `")))))"`.
3. (bonus : à faire plus tard) Ajouter les opérations de comparaison `"="`, `"<"` et `">"` ainsi que les opérateurs booléens `"&"`, `"|"` et `"~"` correspondant respectivement aux opérations `and`, `or` et `not`. Ces opérations sont moins prioritaires que les opérations de comparaison, elle même moins prioritaires que les opérations arithmétiques. Il faudra penser à ajuster le nombre de parenthèses associé à `"("` et `")"`. Remarque : chaque opération ne pouvant utiliser qu'un symbole, `"a<=b"` sera codé `"(a<b | a=b)"`.

Exercice 4 — Parenthésage

On suppose que le dictionnaire créé lors de l'exercice précédent et accessible comme variable globale.

Écrire une fonction `parenthèsage(formule)` transformant la chaîne `formule` en appliquant ces trois étapes :

1. on ajoute un symbole `'('` au début de la formule et `')'` à la fin de la formule ;
2. on supprime les espaces et on remplace les négations par des symboles `'_'` (*cf.* Exercice 2) ;
3. on remplace chaque symbole qui correspond à une clé du dictionnaire par la valeur correspondante.

Prenons comme exemple la chaîne `"-1+2 * 3"`

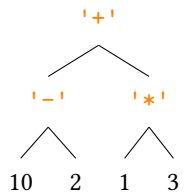
- après l'étape 1 : `"(-1+2 * 3)"`
- après l'étape 2 : `"(_1+2*3)"`
- après l'étape 3 : `"((((_1)))+(((2))*((3))))"`
  - on a remplacé `"("` par `"(((("` et `")"` par `"))))"`
  - on a remplacé `"+"` par `")))+((("` et `"*"` par `"))*(("`

```
>>> parenthèsage("-1+2 *3")
'((((_1)))+(((2))*((3))))'
```

La formule finale est affreuse pour l'œil humain mais a l'avantage de ne nous permettre d'ignorer les règles implicites de priorité entre opérations. Grâce aux nouvelles parenthèses, l'ordre des calculs est parfaitement défini (même si de nombreuses parenthèses sont clairement inutiles).

Exercice 5 — À la recherche de la racine

Lorsque plusieurs opérations de même priorité s'enchaînent, l'opération principale (celle correspondant à la racine de l'arbre) est la dernière. En effet la formule $10-2+3$ doit s'interpréter $(10-2)+3$ et non $10-(2+3)$. Malheureusement, on ne peut pas se contenter de renvoyer la dernière opération de la chaîne : il faut tenir compte des parenthèses. Par exemple, dans la formule $10-2+(1*3)$, l'opération principale est l'addition et l'arbre correspondant sera celui ci-dessous.



1. Pour trouver la racine, on va parcourir la chaîne de droite à gauche (à l'envers donc). On se donne une variable `niveau`, initialisée à `0`, correspondant au niveau de profondeur dans les parenthèses. Si le caractère lu est `")"`, on incrémente `niveau` et si le caractère lu est `"("` on décrémente `niveau`.
   Pour cela, écrivez une fonction `indice_racine(chaîne)` Si en parcourant la chaîne on tombe sur une opération (un des symboles de la chaîne `"+-*/^<=>|&~"`) pendant que la variable `niveau` vaut `0`, on renverra l'indice correspondant. Cela signifie que l'opération est la racine de l'arbre. Si une telle opération n'est pas trouvée on renverra `None`.
2. En déduire une fonction `découpage_formule(chaîne)` qui renvoie un triplet de trois chaînes correspondant à la partie gauche du calcul, à l'opération principale et à la partie droite du calcul. On pourra utiliser l'indice renvoyé par la fonction précédente ainsi que la fonction `découpage` du premier exercice. Si l'indice de la racine est `None`, on renverra alors `None`

```
>>> découpage_formule("1-5+2-(3*5)")
('1-5+2', '-', '(3*5)')
```

Exercice 6 — Arborisons le calcul

> Pour manipuler les arbres, vous ne pouvez utiliser que les fonctions suivantes :
> — `arbre(r,Ag,Ad)` renvoie un arbre de racine `r` et de fils `Ag` (gauche) et `Ad` (droit) ;
> — `est_feuille(A)` renvoie `True` si `A` est une feuille, et `False` sinon ;
> — `racine(A)` renvoie la racine de l'arbre `A` : `'+'`, `'-'`, `'*'`, `'/'`, ou `'^'` ;
> — `fg(A)` renvoie le fils gauche de `A` et `fd(A)` renvoie le fils droit de `A`.
> https://upinfo.univ-cotedazur.fr/~obaldellon/L1/bi2/tp3/abe.py

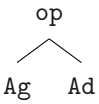
Nous avons grâce à l'exercice 4 une formule correctement parenthésée. Il reste maintenant à la transformer en un arbre. L'objectif est d'écrire une fonction récursive qui à partir d'une chaîne `formule` renvoie un tel arbre.

L'algorithme est récursif et crée un arbre. Mais attention, ce n'est pas une récurrence sur un arbre mais sur une chaîne. Prenons par exemple la chaîne `"1-5+2-(3*5)"`. On la découpe en trois parties : `"1-5+2"` correspondant au calcul du sous-arbre gauche, `'-'` à la racine et `"(3*5)"` au calcul du sous-arbre droit. L'arbre obtenu sera :

où `Ag` et `Ad` sont les sous-arbres, calculés à partir de `"1-5+2"` (pour `Ag`) et `"(3*5)"` (pour `Ad`).

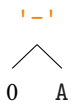
On se donne comme argument une chaîne de caractères `formule`. L'algorithme commence par y appliquer la fonction `découpage_formule` de l'exercice précédent. Selon le résultat, deux cas se présentent :

Premier cas, le découpage renvoie un triplet `(gauche, op, droite)`. On transforme récursivement `gauche` et `droite` en deux sous arbres `Ag` et `Ad` et on renvoie l'arbre :



Second cas, le découpage renvoie `None` Dans ce cas on regarde le premier caractère de la formule.

- Si le premier caractère de `formule` est un chiffre, cela signifie que la chaîne représente un entier et dans ce cas, il suffit de la convertir en feuille avec la fonction de conversion `int`.
- Si le premier caractère de `formule` est une lettre, cela signifie que la chaîne représente une variable et dans ce cas, il n'y a rien à faire (une variable étant codée naturellement sous forme de chaîne). On pourra utiliser la méthode `"a".isalpha()` pour tester si un caractère est une lettre.
- Si le premier caractère de `formule` est une parenthèse ouvrante `"("`, cela signifie que la formule est entourée de parenthèses inutiles, dans ce cas, on recommence récursivement avec la chaîne `intérieur(formule)` (*cf.* Exercice 1).
- Si le premier caractère est un `"_"`, dans ce cas, en notant « `A` » l'arbre obtenu en arborisant récursivement la chaîne `suivant(formule)` (*cf.* Exercice 1), on renvoie l'arbre de la forme :

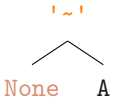


— Sinon, on lance une `ValueError` avec un petit message explicatif.

1. Écrire une première fonction `arboriser_propre(ch)` qui calcul l'arbre correspondant à la formule en supposant que la formule est proprement parenthésée.
2. En déduire la fonction `arboriser(ch)` qui transorme une formule classique en arbre. Il suffira de faire appel à la fonction précédente ainsi qu'à la fonction `parenthèsage`

Exercice 7 — Évaluons une expression

1. On appelle arbre arithmétique un arbre dont les feuilles sont des entiers (sans variables donc). Écrire une fonction `est_arithmétique(A)` qui renvoie `True` si `A` est arithmétique et `False` sinon.
2. Écrire une fonction `calculer_arbre(A)` qui renvoie la valeur de l'expression arithmétique (on suppose que `A` est arithmétique, on lancera une `ValueError` sinon).
3. En déduire une fonction `évaluer(chaîne)` qui calcule la valeur de la chaîne donnée en paramètre.
4. Implémenter les autres opérations mentionnées à la question 3 de l'**Exercice 3** . La négation sera codé par un sous-arbre :



Exercice 8 — Ajout de l'environnement

1. Écrire une fonction `variables(A)` qui renvoie la liste des variables d'un arbre `A`.
2. On appelle environnement un dictionnaire qui associe un entier pour chaque variables de `A`. Écrire une fonction `évaluer_environnement(A,env)` qui évalue l'arbre `A` dans l'environnement `env`.

Exercice 9 — Logique et tables de vérité

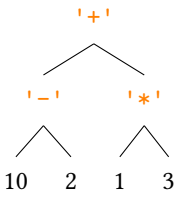
Cet exercice permet de montrer comment démontrer automatiquement certain théorème mathématique.

1. Écrire une fonction `table_de_vérité(formule)` qui affiche la table de vérité associée à une formule donnée sous forme de chaîne.
2. On appelle tautologie une formule qui est toujours vraie. On appelle de même contradiction une formule toujours fausse. Écrire les fonctions `tautologie(formule)` et `contradiction(formule)`
3. On dit que deux formules sont équivalentes si elles ont la même table de vérité.
   Écrire la fonction `équivalence(formule1, formule2)` correspondante.

Exercice 10 — Calcul formel et dérivée

Nous souhaitons maintenant utiliser la structure d'arbre pour faire du calcul automatique sur les formules.

1. Écrire une fonction `afficher(arbre)` qui affiche sous forme infixe la formule correspondante à l'arbre. Par exemple pour l'arbre suivant.



2. `dériver(formule)`