

TP/projet Théorie des Langages 1

Implémentation des automates

L'objectif de ce TP est d'implémenter un automate qui reconnaît un sous-ensemble des nombres flottants acceptés par Python 3, langage que vous avez déjà abordé dans l'exercice 13 du recueil de TD. Le projet prolongera ce travail en reconnaissant les expressions arithmétiques sur les nombres flottants.

Le projet est à rendre pour le **vendredi 8 décembre 2023**.
Le TP et le projet comptent ensemble pour 25% de la note de la matière.

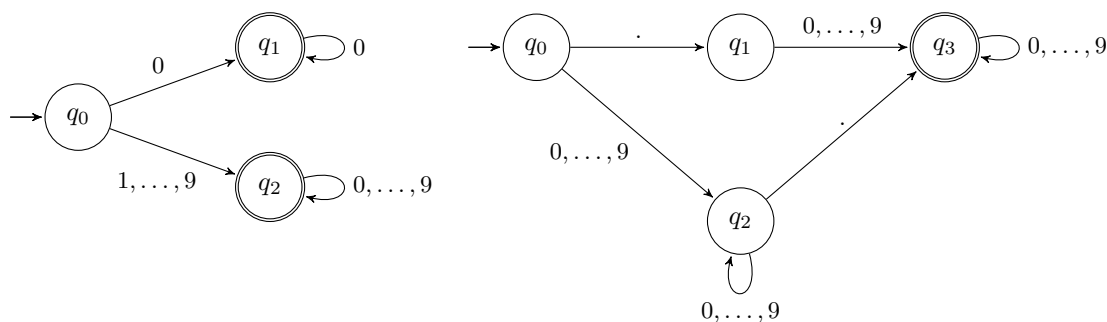
Le vocabulaire V est fixé à

$$V \stackrel{\text{def}}{=} \{0, \dots, 9, \text{e}, \text{E}, ., +, -\}.$$


On définit les langages suivants :

$$\begin{aligned}\text{nonzerodigit} &\stackrel{\text{def}}{=} \{1, \dots, 9\} \\ \text{digit} &\stackrel{\text{def}}{=} \{0\} \cup \text{nonzerodigit} \\ \text{integer} &\stackrel{\text{def}}{=} \text{nonzerodigit}(\text{digit}^*) \cup \{0\}^+ \\ \text{dot} &\stackrel{\text{def}}{=} \{.\} \\ \text{pointfloat} &\stackrel{\text{def}}{=} (\text{digit}^+)\text{dot}(\text{digit}^*) \cup \text{dot}(\text{digit}^+)\end{aligned}$$

Voici des automates déterministes (mais non complets) qui reconnaissent les langages `integer` et `pointfloat`.



Vous allez programmer en Python ces automates en utilisant la méthode des fonctions vue en cours. L'entrée est supposée se terminer par un caractère spécial `END` (appelée *sentinelle de fin d'entrée*) qui n'apparaît pas dans V (donc pas dans les mots à reconnaître) et qui sera par exemple le caractère de fin de ligne, noté `\n` (en Python, `'\n'` est bien une chaîne d'un seul caractère : le caractère de fin de ligne). Ainsi, nos automates seront en fait définis sur le langage $V_E = V \cup \{ \text{END} \}$ et l'automate pour `integer` reconnaîtra en fait le langage `integer.END`. Comme vu en cours, ceci permet de lire les symboles de l'entrée un à un (on parle alors de programme *en ligne*) à l'aide de la fonction `next_char` sans avoir besoin de connaître la longueur de l'entrée. Dans votre code Python, utilisez bien la variable `END` (et non la chaîne de caractères `"END"` ou `'END'`) plutôt que `'\n'`, sinon le script de test de la question 3 ne fonctionnera pas.

▷ **QUESTION 0** Récupérez le squelette du TP et le fichier de test (les fichiers `tp.py` et `test_tp.py`) depuis le répertoire `Documents/TP/` de la page Chamilo du cours. Sauvegardez-les localement à l'aide de l'icône .

▷ QUESTION 1 Dans le fichier `tp.py` fourni, que font les fonctions `nonzerodigit` et `digit` ? Pensez à les utiliser dans la suite. Quelle vérification effectue la fonction `next_char` ?

Méthode des fonctions :

- chaque état est une fonction qui effectue une transition et appelle directement la fonction de l'état suivant ;
- si le caractère lu est `END`, on renvoie `True` ou `False` suivant que l'état est acceptant ou non.

▷ QUESTION 2 En utilisant la méthode des fonctions, programmez les automates pour les langages `integer` et `pointfloat` (fonctions `integer_Q2` et `pointfloat_Q2`). Lorsque l'entrée est un mot du langage $V^*.\{\text{END}\}$, les automates devront retourner sans erreur soit `True`, soit `False`. En particulier, veillez à bien renvoyer une valeur dans tous les chemins d'exécution possibles de vos fonctions.

Lorsque le mot d'entrée contient un préfixe dans V^* qui n'est le préfixe d'aucun mot reconnu, on autorise les automates à retourner `False` de façon anticipée (donc sans avoir lu tout le mot). Par exemple pour `pointfloat`, le mot `1a2` va systématiquement provoquer une erreur (car `a` $\notin V$), tandis que `ea` peut soit retourner `False` (après avoir lu seulement `e`), soit provoquer une erreur (après avoir lu `a`).

▷ QUESTION 3 Testez vos implémentations avec quelques petits exemples en exécutant directement `tp.py`. Pour `pointfloat_Q2`, il faudra changer la fonction à tester à la fin du fichier `tp.py`.

Puis examinez le script fourni `test_tp.py` : il va servir à valider vos différents automates de façon plus exhaustive. Pour cela, il faut indiquer en début du script quels automates doivent être testés. Validez vos implémentations de la question 2 avec `test_tp.py`. Pour déboguer vos fonctions, n'hésitez pas à ajouter des `print` pour observer leur exécution ou à utiliser le débogueur Python.

En pratique, un automate ne sert pas uniquement à reconnaître un mot, on peut vouloir renvoyer ce mot ou faire du calcul avec. Ici, on va renvoyer la valeur du nombre (entier ou flottant) lorsqu'on l'accepte.

▷ QUESTION 4 On veut qu'à tout moment, la valeur du nombre lu jusqu'à présent soit stockée dans une variable globale `int_value`. Notez que `int_value` stocke la *valeur* du nombre et non sa représentation comme une chaîne de caractère. Modifier le *dessin* de l'automate pour `integer` ci-dessus, en ajoutant sur chaque transition les modifications à effectuer pour maintenir dans `int_value` la valeur du nombre lu jusqu'à présent.

▷ QUESTION 5 Modifiez l'implémentation de la fonction `integer_Q2` en une fonction `integer` qui calcule la valeur de l'entier reconnu. Si un mot est accepté, la fonction devra renvoyer le couple (`True`, `int_value`). Si un mot est rejeté, elle devra renvoyer le couple (`False`, `None`). Elle pourra toujours provoquer une erreur lorsqu'un caractère hors de $V \cup \{\text{END}\}$ est rencontré.

▷ QUESTION 6 Pour pouvoir tester votre fonction sur de petits exemples, vous allez devoir modifier le code de test à la fin du fichier `tp.py`. En effet, les automates précédents renvoyaient `True` ou `False` alors que désormais ils renvoient (`True`, `int_value`) ou (`False`, `None`). Après avoir lu et compris le code de test en fin du fichier `tp.py`, commentez et décommentez les bonnes lignes dans ce code, puis tester votre fonction sur quelques petits exemples. Validez votre fonction avec `test_tp.py` (cf. explications en question 3).

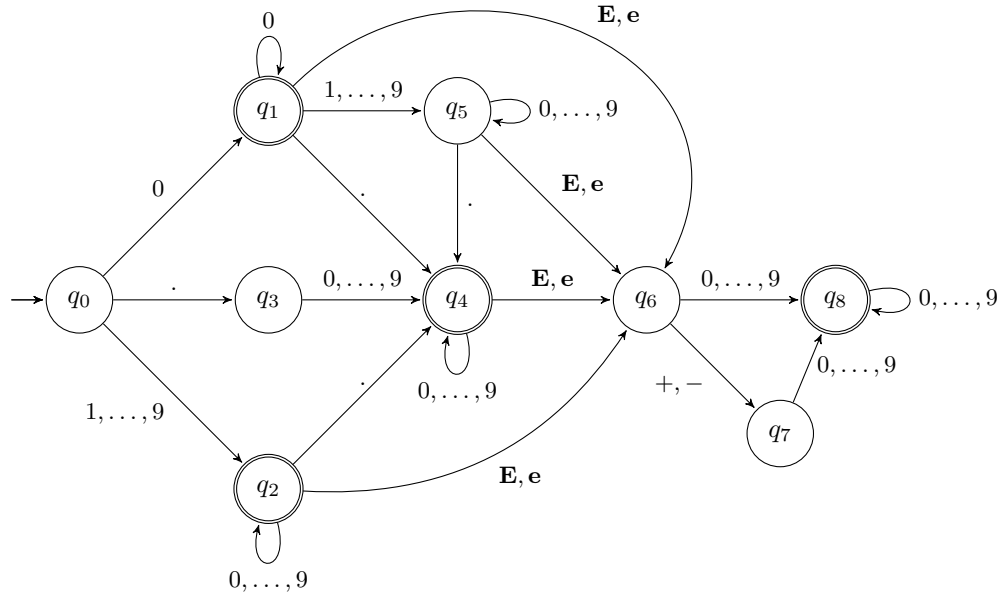
▷ QUESTION 7 Refaites les questions 4 et 5 pour le langage `pointfloat`. Le calcul de la valeur s'effectuera à l'aide de deux variables : la variable `int_value` comme précédemment et une nouvelle variable `exp_value` qui déterminera « où placer la virgule », c.-à-d. de combien de puissance de 10 décaler cet entier pour obtenir le nombre flottant attendu. Autrement dit, le résultat sera `int_value` $\times 10^{-\text{exp_value}}$. Validez votre implémentation sur de petits exemples puis avec `test_tp.py`.

Voici quelques exemples de nombres flottants à reconnaître : 45.3, 606., .345.

▷ QUESTION 8 Pour chacun des langages suivants, donnez puis implémentez un automate déterministe qui renvoie la valeur calculée : la valeur de l'exposant pour `exponent`, la valeur de l'entier flottant pour `exponentfloat`, la valeur du nombre pour `number`. Pour faciliter le calcul dans `exponentfloat`, on introduit une nouvelle variable `sign_value` qui vaut 1 ou -1 suivant que l'exposant est positif ou négatif.

$$\begin{aligned}
\text{exponent} &\stackrel{\text{def}}{=} \{\mathbf{e}, \mathbf{E}\} \{\varepsilon, +, -\} \text{digit}^+ \\
\text{exponentfloat} &\stackrel{\text{def}}{=} (\text{digit}^+ \cup \text{pointfloat}) \text{exponent} \\
\text{number} &\stackrel{\text{def}}{=} \text{integer} \cup \text{pointfloat} \cup \text{exponentfloat}
\end{aligned}$$

Pour vous aider, voici un automate pour **number** (déterministe mais non complet) :



Projet : reconnaissance des expressions arithmétiques

En utilisant les automates précédents, nous allons reconnaître un langage (un peu) plus complexe : les expressions arithmétiques sur les quatre opérations.

Pour cela, on ajoute au vocabulaire les parenthèses et des symboles pour les opérations :

$$V \stackrel{\text{def}}{=} \{\mathbf{0}, \dots, \mathbf{9}, \mathbf{e}, \mathbf{E}, ., +, -, *, /, (,), \text{ } \}.$$

Remarque : le dernier caractère est une espace, qui est nécessaire pour séparer des nombres consécutifs.

Calculatrice préfixe

L'objectif de cette partie est de programmer en Python une calculatrice *en notation préfixe*¹. Le principe d'une telle notation est que chaque opérateur est d'arité fixe (c.-à-d. a un nombre d'arguments fixe) et qu'il est placé syntaxiquement en position préfixe, c'est-à-dire avant ses arguments. Ces contraintes suffisent à rendre les expressions du langage non-ambiguës sans recourir à des parenthèses.

1. Aussi appelée « notation polonaise », car inventée par le logicien polonais J. Łukasiewicz en 1924.

Concrètement, la syntaxe de cette calcullette se décrit à l'aide de la grammaire non-ambiguë suivante :

$$\begin{aligned}\underline{exp} &\rightarrow \text{number} \\ \underline{exp} &\rightarrow + \underline{exp} \underline{exp} \\ \underline{exp} &\rightarrow - \underline{exp} \underline{exp} \\ \underline{exp} &\rightarrow * \underline{exp} \underline{exp} \\ \underline{exp} &\rightarrow / \underline{exp} \underline{exp}\end{aligned}$$

En attendant de voir en détail les grammaires dans le cours, vous pouvez considérer cela comme une définition de langage par induction structurale. Ici, **number** est le cas de base et les règles $\underline{exp} \rightarrow \odot \underline{exp} \underline{exp}$ pour $\odot \in \{+, -, *, /\}$ sont les quatre cas inductifs.

▷ QUESTION 9 Est-ce que ce langage est régulier ? hors-contexte² ? Justifier.

Pour traduire notre grammaire en un programme calculant le résultat d'une expression arithmétique en notation préfixe, on va étendre la méthode des fonctions. Chaque non-terminal \underline{X} de notre grammaire (ici \underline{exp}) se traduit par une fonction **eval_X** qui va calculer la valeur associée à notre expression, par induction structurale, c.-à-d. de façon potentiellement récursive. Pour cela, il faudra tout d'abord identifier quelle règle de la grammaire est utilisée à l'aide du premier symbole rencontré. Puis, dans le cas de base, utiliser le programme précédent pour renvoyer la valeur de **number**. Pour les autres règles, obtenir les valeurs des sous-expressions par des appels récursifs et calculer la valeur de l'expression à l'aide de l'opération utilisée (addition, soustraction, multiplication, division). Par exemple, pour le cas $+ \underline{exp} \underline{exp}$, on aura le fragment de programme suivant :

```
ch = next_char()
if ch == '+':
    n1 = eval_exp()
    n2 = eval_exp()
    return n1 + n2
```

▷ QUESTION 10 Écrire la fonction **eval_exp** qui calcule la valeur d'une expression arithmétique en notation préfixe.

▷ QUESTION 11 Tester votre fonction **eval_exp** sur l'entrée « + 13 12 ». Pensez à commenter/décommenter la bonne ligne dans la fonction de test, car notre fonction calcule une valeur mais ne renvoie pas **True** ou **False** (cf. question 6). Quel problème constatez-vous ? À quel moment et pourquoi se produit l'erreur ?

Modifiez **number** en autorisant une espace comme fin de mot, en plus de **END**. La fonction **eval_exp** renvoie alors un résultat mais quel autre problème observe-t-on ?

Afin de régler ces problèmes, une solution est d'autoriser notre programme et nos automates précédents à lire un caractère en avance *sans le consommer* et de chercher à reconnaître *le plus long préfixe* de l'entrée qui peut être reconnu sans erreur. On ajoute donc une fonction **peek_char** qui fait précisément cela : elle renvoie le prochain caractère sans avancer dans l'entrée. Pour consommer le prochain caractère, il faudra faire appel à la fonction **consume_char**. Notez que **next_char** peut alors se définir comme la composition de **peek_char** et **consume_char**.

Attention : ne plus utiliser **next_char** dans la suite mais seulement **peek_char** et **consume_char**.

▷ QUESTION 12 Modifiez les fonctions **number** et **eval_exp** en **number_v2** et **eval_exp_v2** en utilisant **peek_char** pour régler les problèmes précédents. Assurez-vous que la fonction **eval_exp_v2** fonctionne correctement cette fois-ci sur l'entrée « + 13 12 ».

2. Vous verrez cette notion au cours 9, attendez-le pour répondre à cette question.

Calculatrice infixé

Revenant à une notation plus usuelle, on définit les expressions arithmétiques par la grammaire G_1 suivante. Comme nous allons le voir, cette grammaire est plus complexe et nous ne serons pas capables de calculer efficacement la valeur d'une expression, il faudra attendre le cours de théorie des langages 2 (TL2) pour ce faire.

$$\begin{aligned} \underline{exp} &\rightarrow \text{number} \mid \underline{exp} \underline{op} \underline{exp} \mid (\underline{exp}) \\ \underline{op} &\rightarrow + \mid - \mid * \mid / \end{aligned}$$

On définit un nouveau langage pour les opérateurs :

$$\text{operator} \stackrel{\text{def}}{=} \mathcal{L}(\underline{op}) = \{+, -, *, /\}$$

▷ QUESTION 13 Montrer que le langage $\mathcal{L}(\underline{exp})$ n'est pas régulier.

N'étant pas régulier, on ne peut reconnaître ce langage par un automate. En revanche, afin de simplifier le traitement global, on utilise un automate pour *segmenter* un texte, c.-à-d. séparer les mots, ou *lexèmes* les uns des autres. Un lexème est un fragment de l'expression qui ne peut être séparée d'avantage sans modifier le sens de l'expression. De manière équivalente, on ne peut ajouter d'espace au milieu du fragment sans en modifier le sens. Par exemple, « 13 » (sans les guillemets) est un lexème : en le séparant en 1 et 3, on en modifie le sens (l'entier 13 d'un côté, les entiers 1 et 3 de l'autre). Par contre, « (65 » n'en est pas un : on peut le séparer en (et 65 sans en changer le sens. Ici, l'ensemble des lexèmes est $\text{Lex} \stackrel{\text{def}}{=} \text{number} \cup \text{operator} \cup \{ (,) \}$.

▷ QUESTION 14 Donnez et programmez un automate qui reconnaît le langage **Lex**.

Utiliser des lexèmes permet de s'abstraire de certains détails de représentation, par exemple la base utilisée pour représenter un entier. En langue naturelle, cela reviendrait à vérifier l'orthographe (tous les mots utilisés existent) avant la grammaire (les phrases sont bien formées).

Dans un compilateur, le traitement d'un programme se fait en plusieurs phases dont les deux premières sont l'analyse lexicale puis l'analyse syntaxique. Vous verrez cela plus en détail dans le cours de TL2. Pour illustrer cela, prenons un exemple en français : l'analyse lexicale découperait le texte « **Le chat mange la souris** » en « Article, Nom, Verbe, Article, Nom ». L'analyse syntaxique permettrait ensuite de regrouper Article et Nom en GN (Groupe Nominal), puis remarquerait que le second GN est un COD, ce qui avec un verbe forme un GV (groupe verbal), et enfin identifierait la phrase comme GN puis GV.

Revenons à notre exemple d'expressions arithmétiques. Ici, outre reconnaître le langage **Lex**, on peut aussi vouloir produire une version plus abstraite du texte d'entrée, dans laquelle chaque lexème est remplacée par sa « catégorie grammaticale », possiblement avec une valeur.

Voici les catégories grammaticales ou *tokens* que nous allons utiliser : **NUM** pour les nombres, **ADD** pour « + », **SOUS** pour « - », **MUL** pour « * », **DIV** pour « / », **OPAR** pour « (», **FPAR** pour «) ». Le token **NUM** est le seul qui possède une valeur qui indique la valeur du nombre reconnu.

Ainsi, le texte « (3) + 4 * 5 » sera transformé en la suite de tokens **OPAR**, **NUM**(3), **FPAR**, **ADD**, **NUM**(4), **MUL**, **NUM**(5).

▷ QUESTION 15 Modifier le programme de la question précédente pour qu'il renvoie à chaque fois le token correspondant au lexème reconnu. Pour comprendre quel est le token correspondant, il faut se référer à son codage comme un entier qui correspond à leur ordre de définition : **NUM**, **ADD**, **SOUS**, **MUL**, **DIV**, **OPAR**, **FPAR**. Ainsi, 0 correspond à **NUM**, 3 à **MUL** et 6 à **FPAR**.

Dans la grammaire G_1 , l'expression $3 + 4 * 5$ est ambiguë : on peut la comprendre comme $(3 + 4) * 5$ ou bien $3 + (4 * 5)$. Afin d'éviter trop de parenthèses, on introduit des priorités : multiplication et division ont priorité sur addition et soustraction et en cas d'égalité de priorité, les expressions sont implicitement parenthésées à gauche. Ainsi, $3 + 4 * 5$ signifie $3 + (4 * 5)$, $4 - 5 + 6$ signifie $(4 - 5) + 6$ (car $-$ et $+$ ont même priorité).

▷ QUESTION 16 Donnez une grammaire G_2 équivalente à G_1 et qui en lève les ambiguïtés en utilisant les priorités suivantes :

Priorité 0 (la plus forte) $\frac{\underline{exp}}{\underline{exp}} \rightarrow \mathbf{number}$
 $\frac{\underline{exp}}{\underline{exp}} \rightarrow \mathbf{OPAR} \underline{exp} \mathbf{FPAR}$

Priorité 1 $\frac{\underline{exp}}{\underline{exp}} \rightarrow \underline{exp} \mathbf{MUL} \underline{exp}$
 $\frac{\underline{exp}}{\underline{exp}} \rightarrow \underline{exp} \mathbf{DIV} \underline{exp}$

Priorité 2 (la plus faible) $\frac{\underline{exp}}{\underline{exp}} \rightarrow \underline{exp} \mathbf{ADD} \underline{exp}$
 $\frac{\underline{exp}}{\underline{exp}} \rightarrow \underline{exp} \mathbf{SOUS} \underline{exp}$

Contrairement au cas préfixe, lire le prochain token n'est pas suffisant pour savoir quelle règle est utilisée. Par exemple, toute expression commence ou bien par une parenthèse ouvrante, ou bien par un nombre. Dans le cas d'un nombre, on ne sait pas si l'expression entière est ce nombre (règle $\underline{exp} \rightarrow \mathbf{number}$) ou si c'est le début d'une opération (règle $\underline{exp} \rightarrow \underline{exp} \odot \underline{exp}$). On ne peut donc pas construire directement de fonction d'évaluation comme pour les expressions en notation préfixe.

La solution que vous verrez en TL2 sera de transformer cette grammaire en une grammaire équivalente LL(1), c.-à-d. pour laquelle on saura quelle règle utiliser en regardant uniquement le prochain token.