Projet Théorie des Langages

Analyse

L'objectif de ce projet est d'implémenter une calculatrice sur des nombres entiers, flottants et en notation scientifique qui connaît les quatre opérations arithmétiques usuelles, les puissances et la factorielle. Elle traite une suite de « calculs », constitués d'une expression suivie par le caractère « ; ». Un calcul peut faire appel aux vaeurs des calculs précédents, par la syntaxe « $\#i$ » qui fait référence au i^{e} calcul de la suite.
Le résultat est la liste complète des valeurs des calculs (ou un message d'erreur le cas échéant). Par exemple, la suite de calculs `3 + 2 * 2 ; #1 / 2 + 3 ; #1 + #2 ;` produira successivement : 7 ; 6,5 et 13,5. Le résultat final sera donc la liste `[7, 6.5, 13.5]`.

> Le projet est à rendre pour le **dimanche 7 décembre 2025**.
> Le projet compte pour 20% de la note de la matière.

Vous devez respecter les noms de fonctions donnés et le format des résultats attendus car l'évaluation utilisera notamment des tests automatiques.

1 Vue d'ensemble et état initial

La calculatrice est constituée de deux fonctions majeures :
— un *analyseur lexical* : il consomme une séquence de caractères du flot d'entrée et produit un *token*.
— un *analyseur grammatical* : il appelle quand il le faut le précédent pour obtenir un par un les tokens de la suite, afin de vérifier que celle-ci constitue bien la frondaison d'un arbre d'analyse, pour une grammaire hors-contexte définissant les suites de calculs. [1]

La construction du résultat (ici la liste des valeurs des calculs) s'effectue au cours de l'analyse grammaticale en implémentant le calcul d'attributs associé à la grammaire.

L'ensemble des caractères d'entrée forme l'alphabet d'entrée V_C :

$$V_C \stackrel{\text{def}}{=} \{\mathbf{0}, \ldots, \mathbf{9}, \mathbf{e}, \mathbf{E}, ., +, -, *, /, \hat{}, !, (,), \#, ;\}.$$

L'ensemble des tokens forme le vocabulaire terminal V_T :

$$V_T \stackrel{\text{def}}{=} \{\mathbf{NUM}, \mathbf{ADD}, \mathbf{SUB}, \mathbf{MUL}, \mathbf{DIV}, \mathbf{POW}, \mathbf{FACT}, \mathbf{OPAR}, \mathbf{CPAR}, \mathbf{CALC}, \mathbf{SEQ}, \mathbf{END}\}.$$

Enfin, le vocabulaire non terminal de la grammaire de notre calculatrice est $V_N \stackrel{\text{def}}{=} \{\mathsf{exp}, \mathsf{input}\}$.

Voici les différents fichiers présents dans l'archive fournie :

definitions.py Définit les tokens et les lexèmes.

lexer.py Contient l'analyse lexicale (à vous de le compléter)

parser.py Contient l'analyse grammaticale (à vous de le compléter)

test_lexer.py Tests pour votre lexer

test_parser.py Tests pour votre parser

test_calc.py Tests pour votre parser avec attributs

▷ **Question 0.** Récupérez le squelette du projet contenant les fichiers précédents depuis le répertoire `Documents/projet/` de la page Chamilo du cours. Sauvegardez-les localement à l'aide de l'icône 💾.

1. Le couple des deux analyseurs constitue un *analyseur syntaxique*.

2 Analyse lexicale

L'analyse lexicale se fera avec des automates. Outre les caractères de V_C, les automates pourront lire des caractères blancs (espaces, tabulations, sauts de ligne), qui pourront servir de séparateurs entre les lexèmes, ainsi qu'une *sentinelle de fin d'entrée* : un caractère spécial `EOI` (pour End Of Input) qui n'apparaît pas dans V_C (donc pas dans les mots à reconnaître) et qui sera par exemple le caractère de saut de ligne, noté `\n` (en Python, `'\n'` est une chaîne d'un seul caractère : le saut de ligne).

Vous ne devez pas supposer que la valeur de `EOI` est `\n`, utilisez à la place directement la variable `EOI` dans `lexer.py`. En effet, si dans les tests en ligne de commande, `EOI` vaudra bien `\n`, ce ne sera plus le cas avec des fichiers de test.

Le vocabulaire V pour ces automates est ainsi fixé à

$$V \stackrel{\text{def}}{=} V_C \cup \text{SEP} \cup \{\texttt{EOI}\} \qquad \text{avec} \qquad \text{SEP} \stackrel{\text{def}}{=} \{\ \texttt{␣},\ \texttt{\t},\ \texttt{\n}\} \setminus \{\texttt{EOI}\}$$

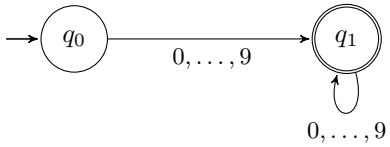
où ␣ représente une espace. On s'assure de retirer `EOI` de SEP pour éviter la possible confusion sur le caractère `\n`.

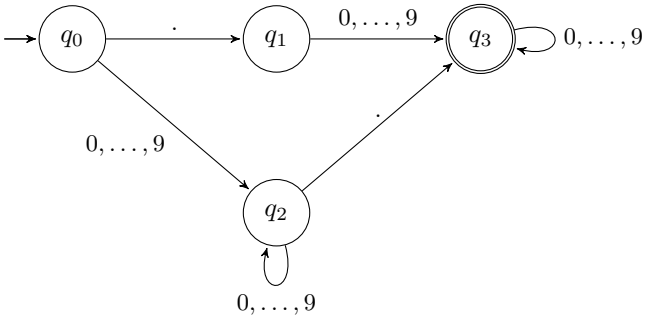
On définit les langages suivants :

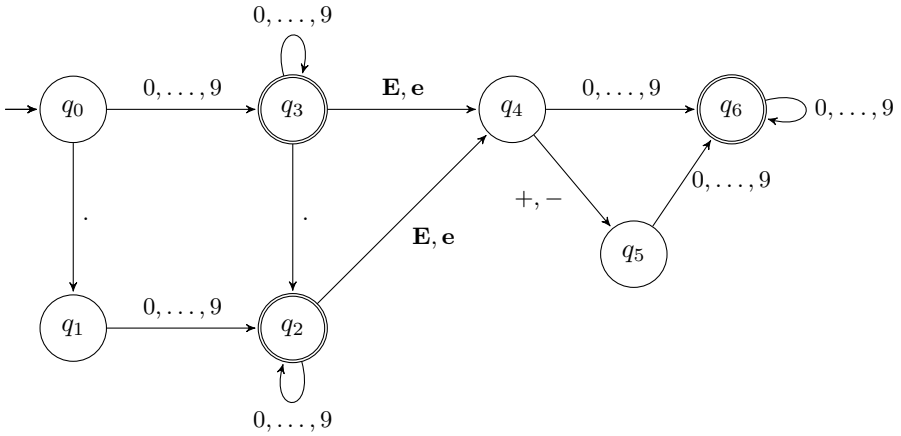
$$\begin{aligned}
\texttt{digit} &\stackrel{\text{def}}{=} \{\mathbf{0}, \ldots, \mathbf{9}\} \\
\texttt{dot} &\stackrel{\text{def}}{=} \{.\} \\
\texttt{integer} &\stackrel{\text{def}}{=} \texttt{digit}^+ \\
\texttt{pointfloat} &\stackrel{\text{def}}{=} \texttt{integer}\,\texttt{dot}\,(\texttt{digit}^*) \ \cup\ \texttt{dot}\,\texttt{integer} \\
\texttt{exponent} &\stackrel{\text{def}}{=} \{\mathbf{e}, \mathbf{E}\}\,\{\varepsilon, +, -\}\,\texttt{integer} \\
\texttt{number} &\stackrel{\text{def}}{=} (\texttt{integer} \cup \texttt{pointfloat})\,(\texttt{exponent} \cup \{\varepsilon\})
\end{aligned}$$

La terminologie est tirée de la documentation de Python. En particulier, ici et dans toute la suite, `integer`, « entier »... désignent en fait des *entiers naturels* (non signés). Les entiers négatifs pourront être obtenus par exemple au moyen de l'opérateur unaire « opposé ».

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







Vous allez programmer en Python ces automates en utilisant par exemple l'une des méthodes vues en cours. Ce travail se fera dans le fichier `lexer.py` fourni.

La lecture d'un caractère est découplée de l'avancée dans l'entrée [2] : la fonction `peek_char1` renvoie le prochain caractère de l'entrée et la fonction `consume_char` permet d'avancer d'un caractère et ne renvoie rien (`None`).

▷ **Question 1.** En utilisant les fonctions `peek_char1` et `consume_char` fournies, programmez des fonctions `read_INT_to_EOI` et `read_FLOAT_to_EOI` qui implémentent les automates pour les langages `integer` et `pointfloat`. Ces fonctions doivent renvoyer un booléen qui indique si l'entrée est reconnue ou non. Il faut lire toute l'entrée, c'est à dire jusqu'à atteindre le caractère `EOI`, même s'il est possible de déterminer plus tôt si le mot est accepté ou non.

▷ **Question 2.** Testez vos automates en exécutant le fichier `lexer.py`. Pour cela, vous devez décommenter l'une des trois dernières lignes du fichier `lexer.py` suivant que vous souhaitez tester `read_INT_to_EOI`, `read_FLOAT_to_EOI` ou (plus tard) le lexer complet.

Caractères de prévision Comme expliqué au paragraphe 1, l'analyseur est appelé autant de fois que nécessaire pour produire une suite de tokens correspondant aux lexèmes successifs reconnus. Par conséquent, nos automates doivent reconnaître un *préfixe* de l'entrée complète, donc sans chercher à atteindre le caractère `EOI`. De plus, la *segmentation*, c.-à-d. le découpage des caractères d'entrée en une suite de lexèmes, n'est pas unique dès lors qu'un lexème peut être décomposé en deux lexèmes, ce qui est le cas ici. Par exemple, « #34 » pourrait être reconnu comme un appel au 34^{e} calcul ou comme un appel au 3^{e} calcul suivi du nombre 4. On choisit en général de reconnaître la plus longue séquence de caractères d'entrée qui forme un lexème valide donc ici le 34^{e} calcul et non le 3^{e} calcul suivi du nombre 4. En pratique, cela signifie que l'on arrête la lecture lorsque la suite de l'entrée ne peut donner lieu à un prolongement valide du lexème courant.

Voici quelques exemples :

Texte d'entrée	Premier lexème reconnu
$(3) + 4 * 5$	$($
$+25$	$+$
$1.5e2 + 7$	$1.5e2$
$15ee - 4 + 7$	15
$1.3e + -4 + 7$	1.3

Dans le dernier cas, on voit qu'il faut lire trois caractères d'avance ($e + -$) pour se rendre compte que le « $-$ » suivant le « $+$ » ne donne pas un exposant valide et qu'il faut donc s'arrêter à 1.3.

Ainsi, comme dans une analyse LL(3), on suppose que les automates peuvent accéder aux trois prochains caractères de l'entrée sans les consommer. Ceci se fait avec la fonction `peek_char3`.

Comme on peut voir trois caractères d'avance, il faut que l'entrée se termine par trois sentinelles de fin et non une seule, afin que ces trois caractères soient toujours définis. En pratique, on en ajoute virtuellement une infinité : lorsque les trois prochains caractères sont la sentinelle `EOI`, tout appel à `consume_char` ne fera pas progresser l'entrée et `peek_char3` renverra à nouveau « `EOI EOI EOI` ».

▷ **Question 3.** Outre reconnaître des nombres, on veut calculer leur valeur car elle sera utilisée plus tard dans le calcul de l'expression. Implémentez un automate qui reconnaisse le plus long préfixe de l'entrée dans le langage `integer` et renvoie la valeur entière correspondante. Votre fonction `read_INT` renverra un entier.
Faire de même pour le langage `number` avec la fonction `read_NUM` qui renverra un nombre entier ou flottant. Pour faciliter le calcul, vous pourrez utiliser une variable contenant la valeur de la mantisse (le nombre sans l'exposant, en ignorant le point éventuel), une contenant celle du signe de l'exposant et une contenant la valeur de l'exposant. À vous de déterminer comment mettre à jour ces différentes variables à chaque transition. Par exemple, l'ajout d'un chiffre après la virgule modifiera à la fois la mantisse et l'exposant.

À présent, on souhaite reconnaître un lexème arbitraire de l'entrée et renvoyer le token correspondant ainsi que la valeur associée (nombre pour **NUM** et indice pour **CALC**).

2. Voir la partie « Caractères de prévision » pour la justification de ce choix.

Voici les langages de lexèmes reconnus par chaque token :

$$
\begin{array}{lcl}
\textbf{NUM} & \stackrel{\text{def}}{=} & \texttt{number} \\
\textbf{CALC} & \stackrel{\text{def}}{=} & \{\#\}.\texttt{integer} \\
\textbf{ADD} & \stackrel{\text{def}}{=} & \{+\} \\
\textbf{SUB} & \stackrel{\text{def}}{=} & \{-\} \\
\textbf{MUL} & \stackrel{\text{def}}{=} & \{*\} \\
\textbf{DIV} & \stackrel{\text{def}}{=} & \{/\} \\
\textbf{POW} & \stackrel{\text{def}}{=} & \{\hat{}\} \\
\textbf{FACT} & \stackrel{\text{def}}{=} & \{!\} \\
\textbf{OPAR} & \stackrel{\text{def}}{=} & \{(\} \\
\textbf{CPAR} & \stackrel{\text{def}}{=} & \{)\} \\
\textbf{SEQ} & \stackrel{\text{def}}{=} & \{;\} \\
\textbf{END} & \stackrel{\text{def}}{=} & \{\textbf{EOI}\}
\end{array}
$$

À l'exception de **NUM** et **CALC**, tous ces langages sont des singletons et peuvent être reconnus directement en faisant appel au dictionnaire `TOKEN_MAP` de `definitions.py`.

▷ **Question 4.** En utilisant vos fonctions `read_INT` et `read_NUM` précédentes, programmez la fonction `read_token_after_separators` qui reconnaît le plus long préfixe de l'entrée qui soit un lexème valide. Votre fonction renverra un couple dont la première composante sera un token (un élément de `V_T`) et la seconde sera la valeur de l'attribut de ce token : un nombre flottant pour **NUM**, un nombre entier pour **CALC** et `None` sinon.

Dans l'entrée, les lexèmes successifs peuvent être séparés par des séparateurs, des éléments de SEP.

▷ **Question 5.** Programmez la fonction `next_token` qui reconnaît le prochain token de l'entrée, possiblement précédé par des séparateurs. Formellement, le langage reconnu est $\text{SEP}^*\,.\,\textbf{LEX}$, où **LEX** est l'union des langages de tous les tokens.

▷ **Question 6.** Testez votre lexer en exécutant `lexer.py` avec la dernière ligne décommentée (et les deux précédentes commentées). Vérifiez ensuite votre travail jusqu'à présent en exécutant le fichier de test `test_lexer.py`.

Analyse grammaticale

Voici la grammaire G_1 qui définit le langage engendré par notre calculatrice et le système d'attributs qui calcule la valeur de chaque expression et la liste de ces valeurs.

$$
\begin{array}{lcll}
\mathsf{input}{\uparrow}\ell & \to & \varepsilon & \ell := [\,] \\
 & | & \mathsf{input}{\uparrow}\ell_0\ \mathsf{exp}{\downarrow}\ell_0{\uparrow}n\ \textbf{SEQ} & \ell := \ell_0 \oplus n \\
 & & & \\
\mathsf{exp}{\downarrow}\ell{\uparrow}n & \to & \textbf{NUM}{\uparrow}n & \\
 & | & \textbf{CALC}{\uparrow}i & n := \ell[i-1] \\
 & | & \mathsf{exp}{\downarrow}\ell{\uparrow}n_1\ \textbf{ADD}\ \mathsf{exp}{\downarrow}\ell{\uparrow}n_2 & n := n_1 + n_2 \\
 & | & \mathsf{exp}{\downarrow}\ell{\uparrow}n_1\ \textbf{SUB}\ \mathsf{exp}{\downarrow}\ell{\uparrow}n_2 & n := n_1 - n_2 \\
 & | & \mathsf{exp}{\downarrow}\ell{\uparrow}n_1\ \textbf{MUL}\ \mathsf{exp}{\downarrow}\ell{\uparrow}n_2 & n := n_1 \times n_2 \\
 & | & \mathsf{exp}{\downarrow}\ell{\uparrow}n_1\ \textbf{DIV}\ \mathsf{exp}{\downarrow}\ell{\uparrow}n_2 & n := n_1 \,/\, n_2 \\
 & | & \mathsf{exp}{\downarrow}\ell{\uparrow}n_1\ \textbf{POW}\ \mathsf{exp}{\downarrow}\ell{\uparrow}n_2 & n := n_1^{n_2} \\
 & | & \textbf{SUB}\ \mathsf{exp}{\downarrow}\ell{\uparrow}n_0 & n := -n_0 \\
 & | & \mathsf{exp}{\downarrow}\ell{\uparrow}n_0\ \textbf{FACT} & n := n_0! \\
 & | & \textbf{OPAR}\ \mathsf{exp}{\downarrow}\ell{\uparrow}n\ \textbf{CPAR} &
\end{array}
$$

Vous allez implémenter un reconnaisseur LL(1) pour le langage de cette grammaire. Comme elle est ambiguë, il faut commencer par lever les ambiguïtés et la mettre en forme LL(1)

▷ **Question 7.** (non évaluée) En respectant les priorités données ci-dessous, donnez une grammaire G_2 non ambiguë équivalente à G_1. Pour cette question, inutile d'adapter le calcul d'attributs de G_1 à G_2.

Priorité	Associativité	Règles
0 (la plus forte)	non associative	exp → **NUM**
		exp → **CALC**
		exp → **OPAR** exp **CPAR**
1	associative à droite	exp → exp **POW** exp
2	associative à gauche	exp → exp **FACT**
3	associative à droite	exp → **SUB** exp
4	associative à gauche	exp → exp **MUL** exp
		exp → exp **DIV** exp
5 (la plus faible)	associative à gauche	exp → exp **ADD** exp
		exp → exp **SUB** exp

▷ **Question 8.** (non évaluée) Donnez une grammaire G_3 LL(1) équivalente à G_2. Pour cette question, inutile d'adapter le calcul d'attributs de G_1 à G_3.

▷ **Question 9.** Implémentez la grammaire G_3 en complétant le fichier `parser.py`. Pour chaque non-terminal X, vous aurez une fonction `parse_X` qui analysera un sous-arbre enraciné en X. Pour reconnaître un token *tok*, vous utiliserez la fonction `consume_token(`*tok*`)` qui vérifie que le prochain token est bien *tok* [3] puis le consomme. Pour accéder au prochain token sans le consommer, vous utiliserez la fonction `get_current`.

▷ **Question 10.** Testez manuellement votre grammaire en exécutant le fichier `parser.py`. Une fois que vous êtes satisfait du résultat, lancez le fichier de test `test_parser.py`.

▷ **Question 11.** Adaptez le calcul d'attributs de G_1 à G_2 puis G_3. Ajoutez ce calcul dans votre implémentation de G_3. Pour cela, vous repartirez de la question précédente en copiant le fichier `parser.py` en `calc.py` et vous ajouterez votre calcul d'attributs dans `calc.py`.
Pour le calcul des puissances et de la factorielle, vous pourrez utiliser les fonctions de la bibliothèque `math` : `math.pow` et `math.factorial`. Pour les listes de valeurs des calculs, vous utiliserez des *listes Python*. Voir la doc Python pour l'indexation et la concaténation.

▷ **Question 12.** Testez manuellement votre grammaire en exécutant le fichier `calc.py`. Une fois que vous êtes satisfait du résultat, lancez le fichier de test `test_calc.py`.

Plutôt que de lever une erreur lorsque un token inattendu est vu, on souhaite essayer de rattraper l'erreur. Pour cela, comme vu en cours, on va consommer des tokens jusqu'à obtenir un token sur lequel repositionner la lecture.

▷ **Question 13.** Copiez votre fichier `calc.py` en `rattrapage.py`. Programmez dans `rattrapage.py` une fonction `recover(`*suiv*`)` qui consomme le token courant jusqu'à trouver un token dans *suiv* ou END. Modifiez ensuite vos fonctions `parse_X` pour rattraper les erreurs que votre parser peut déclencher.

3. Elle lève une exception dans le cas contraire