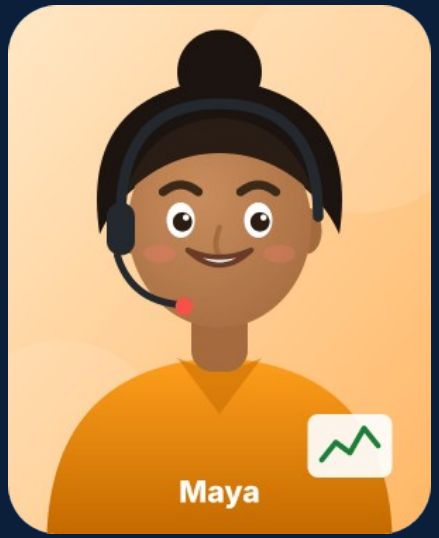
HEF-090 · MISSION 02 / 10

Les jetons et l'analyse lexicale

Avant de comprendre une phrase, il faut la découper en mots. Le lexer transforme un texte brut en une suite de jetons.





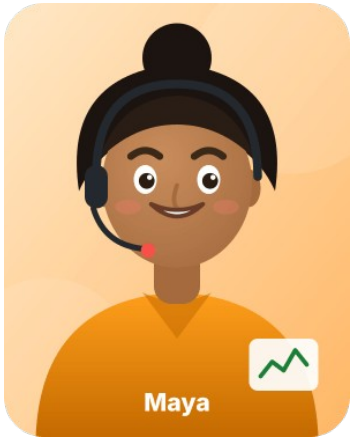
Badge : Découpeur de mots

75 min · intermédiaire

L'histoire

Kodi commence HashScript. Première étape : lire le texte `let prix = 12.5 * quantite; # total`. Il essaie de couper sur les espaces avec `split()`. Ça marche… jusqu'à `prix=12.5*quantite;` sans espaces.

> *« Un lexer ne coupe pas sur les espaces, dit Sam. Il reconnaît des motifs : un nombre, un identifiant, un opérateur. Et il note la ligne et la colonne de chacun, pour les messages d'erreur. »*



Tu écris le premier maillon de la chaîne : le lexer de HashScript.

Ta mission

À la fin, je serai capable de…

1. définir un jeton (genre, texte, position)
2. écrire un lexer avec des expressions régulières
3. gérer mots-clés, identifiants, nombres, opérateurs, commentaires
4. suivre la ligne et la colonne pour les erreurs
5. signaler un caractère inattendu avec sa position

Défi d'échauffement

Découpe en « mots » cette suite de caractères, comme le ferait un programme : `x=3.14*r2; # aire`. Quels sont les morceaux ? Combien de sortes de morceaux différents peux-tu distinguer (nombres, noms, symboles…) ? Que fais-tu de l'espace et du commentaire ?

Le concept

Le jeton (*token*)

Un **jeton** est une unité lexicale : un **genre** (catégorie), un **texte** (le lexème) et une **position**.

Texte	Genre	Position
`let`	MOT (mot-clé)	ligne 1, colonne 1
`prix`	ID (identifiant)	1, 5
`=`	OP	1, 10
`12.5`	NUM	1, 12

Le lexer **ignore** ce qui n'a pas de sens pour la suite (espaces, commentaires).

Le concept

Les motifs avec des expressions régulières

Genre	Motif	Explication
NUM	`\d+(?:\.\d+)?`	des chiffres, éventuellement un point et des chiffres
ID	`[A-Za-z_]\w*`	une lettre ou _, puis lettres, chiffres, _
OP	`` `== ``	!=
COM	`#[^\n]*`	commentaire jusqu'à la fin de ligne
ESPACE	`[ \t\r]+`	à ignorer

Deux règles essentielles

1. **Plus long préfixe d'abord** : <= doit être reconnu avant < ; == avant =. L'ordre des alternatives compte.

Le concept

Deux règles essentielles (suite)

2. **Mot-clé ou identifiant ?** On reconnaît d'abord un identifiant, puis on regarde s'il est dans la liste des mots-clés.

Positions

On suit le **numéro de ligne** et le **début de la ligne courante**. La colonne d'un jeton = `début du jeton − début de la ligne + 1`.

Un lexer en Python

```
import re

MOTS_CLES = {"let", "print", "if", "else", "while", "fun", "return", "true", "fals
   e"}
SPEC = [("NUM", r"\d+(?:\.\d+)?"), ("ID", r"[A-Za-z_]\w*"), ("COM", r"#[^\n]*"),
        ("OP", r"==|!=|<=|>=|[-+*/()<>=;{},]"), ("ESPACE", r"[ \t\r]+"), ("NL", r"
```

Le concept

Un lexer en Python (suite)

```
   \n"), ("ERR", r".")]
MOTIF = re.compile("|".join(f"(?P<{nom}>{motif})" for nom, motif in SPEC))

def tokenize(source):
    ligne, debut_ligne, jetons = 1, 0, []
    for m in MOTIF.finditer(source):
        genre, texte, colonne = m.lastgroup, m.group(), m.start() - debut_ligne +
   1
        if genre == "NL":
            ligne, debut_ligne = ligne + 1, m.end()
        elif genre in ("ESPACE", "COM"):
            continue
        elif genre == "ERR":
            raise SyntaxError(f"caractère inattendu {texte!r} ligne {ligne}, colon
   ne {colonne}")
        else:
            if genre == "ID" and texte in MOTS_CLES:
                genre = "MOT"
            jetons.append((genre, texte, ligne, colonne))
```

Le concept

Un lexer en Python (suite)

```
    jetons.append(("FIN", "", ligne, len(source) - debut_ligne + 1))
    return jetons
```

`m.lastgroup` donne le nom du motif qui a correspondu ; le motif `ERR` (`.`) à la fin récupère tout caractère non reconnu.

Le jeton de fin

On ajoute un jeton spécial `FIN` : le parser peut ainsi toujours « regarder le jeton suivant » sans test spécial de fin de liste.

🧠 ***Astuce** : un lexer est une **machine à états** (automate fini). Les regex en sont une écriture compacte, mais l'idée est la même : lire caractère par caractère et décider de la catégorie.*

Exemple pas à pas

Exemple pas à pas

Problème : tokeniser un programme HashScript et afficher le résultat.

```
import re

MOTS_CLES = {"let", "print", "if", "else", "while", "fun", "return", "true", "fals
   e"}
SPEC = [("NUM", r"\d+(?:\.\d+)?"), ("ID", r"[A-Za-z_]\w*"), ("COM", r"#[^\n]*"),
        ("OP", r"==|!=|<=|>=|[-+*/()<>=;{},]"), ("ESPACE", r"[ \t\r]+"), ("NL", r"
   \n"), ("ERR", r".")]
MOTIF = re.compile("|".join(f"(?P<{nom}>{motif})" for nom, motif in SPEC))

def tokenize(source):
    ligne, debut_ligne, jetons = 1, 0, []
    for m in MOTIF.finditer(source):
        genre, texte, colonne = m.lastgroup, m.group(), m.start() - debut_ligne +
   1
        if genre == "NL":
            ligne, debut_ligne = ligne + 1, m.end()
```

Exemple pas à pas (suite)

```
        elif genre in ("ESPACE", "COM"):
            continue
        elif genre == "ERR":
            raise SyntaxError(f"caractère inattendu {texte!r} ligne {ligne}, colon
   ne {colonne}")
        else:
            if genre == "ID" and texte in MOTS_CLES:
                genre = "MOT"
            jetons.append((genre, texte, ligne, colonne))
    jetons.append(("FIN", "", ligne, len(source) - debut_ligne + 1))
    return jetons

source = "let prix=12.5*qte; # total\nprint prix>=100;"
for jeton in tokenize(source):
    print(jeton)
```

Résultat (extrait) :

Exemple pas à pas

Exemple pas à pas (suite)

```
('MOT', 'let', 1, 1)
('ID', 'prix', 1, 5)
('OP', '=', 1, 9)
('NUM', '12.5', 1, 10)
('OP', '*', 1, 14)
('ID', 'qte', 1, 15)
...
('MOT', 'print', 2, 1)
('ID', 'prix', 2, 7)
('OP', '>=', 2, 11)
```

Lecture :

- `prix=12.5*qte;` est correctement découpé **sans espaces** ;
- le commentaire est ignoré ;
- `>=` est un seul jeton (plus long préfixe) ;

Exemple pas à pas (suite)

- les positions sont exactes (le `print` est ligne 2, colonne 1) ;
- `12.5` est un nombre à virgule : le motif `NUM` essaie de prolonger avec `.chiffres`.

Une erreur lexicale :

```
try:
    tokenize("let a = 5 $ 3;")
except SyntaxError as e:
    print(e)          # caractère inattendu '$' ligne 1, colonne 11
```

À toi, avec aide

Étends le lexer pour reconnaître les **chaînes de caractères** entre guillemets doubles (`"bonjour"`), genre `CHAINE`, sans séquences d'échappement. Teste avec : `print "salut " ;` et un cas d'erreur : une chaîne non terminée `print "abc;` (doit lever une erreur à la bonne position).

À toi, avec aide · réponse

Indices progressifs

Indice 1

Ajoute `("CHAINE", r'"[^"\n]*"')` avant `ERR` dans `SPEC`.

Indice 2

Pour la chaîne non terminée, ajoute un motif `CHAINE_OUVERTE` : `"[^"\n]*` (sans guillemet fermant) placé après `CHAINE`.

Indice 3

Le texte du jeton doit contenir les guillemets ; tu peux les retirer si tu veux la valeur.

Réponse

```
import re

MOTS_CLES = {"let", "print", "if", "else", "while", "fun", "return", "true", "fals
   e"}
SPEC = [("NUM", r"\d+(?:\.\d+)?"), ("ID", r"[A-Za-z_]\w*"), ("COM", r"#[^\n]*"),
        ("CHAINE", r'"[^"\n]*"'), ("CHAINE_OUVERTE", r'"[^"\n]*'),
        ("OP", r"==|!=|<=|>=|[-+*/()<>=;{},]"), ("ESPACE", r"[ \t\r]+"), ("NL", r"
   \n"), ("ERR", r".")]
MOTIF = re.compile("|".join(f"(?P<{nom}>{motif})" for nom, motif in SPEC))

def tokenize(source):
    ligne, debut_ligne, jetons = 1, 0, []
    for m in MOTIF.finditer(source):
        genre, texte, colonne = m.lastgroup, m.group(), m.start() - debut_ligne +
   1
        if genre == "NL":
            ligne, debut_ligne = ligne + 1, m.end()
        elif genre in ("ESPACE", "COM"):
            continue
```

À toi, avec aide · réponse

Réponse (suite)

```
        elif genre == "CHAINE_OUVERTE":
            raise SyntaxError(f"chaîne non terminée ligne {ligne}, colonne {colonn
   e}")
        elif genre == "ERR":
            raise SyntaxError(f"caractère inattendu {texte!r} ligne {ligne}, colon
   ne {colonne}")
        else:
            if genre == "ID" and texte in MOTS_CLES:
                genre = "MOT"
            jetons.append((genre, texte, ligne, colonne))
    jetons.append(("FIN", "", ligne, len(source) - debut_ligne + 1))
    return jetons

print([j[:2] for j in tokenize('print "salut " ;')])
try:
    tokenize('print "abc;')
except SyntaxError as e:
    print(e)          # chaîne non terminée ligne 1, colonne 7
```

À toi, avec aide · réponse

Réponse (suite)

L'ordre compte : `CHAINE` (fermée) est essayée avant `CHAINE_OUVERTE`.

À toi, en solo

Un autre langage. Écris un lexer pour un mini-SQL :

```
SELECT nom, age FROM membres WHERE age >= 18 AND ville = 'Lyon';
```

- mots-clés insensibles à la casse (`select`, `Select`, `SELECT`) : `SELECT FROM WHERE AND OR`, avec leur texte normalisé en majuscules ;
- identifiants, nombres, chaînes entre apostrophes `'Lyon'`, opérateurs `= != < > <= >= , ; ( ) *` ;
- positions ligne/colonne ;
- erreurs : caractère inconnu, chaîne non terminée.

Teste sur au moins 6 requêtes dont 2 invalides.

À toi, en solo · réponse

Pistes de correction

```
import re

MOTS = {"SELECT", "FROM", "WHERE", "AND", "OR"}
SPEC = [("NUM", r"\d+(?:\.\d+)?"), ("ID", r"[A-Za-z_]\w*"), ("CHAINE", r"'[^'\n]*'
   "), ("CHAINE_OUVERTE", r"'[^'\n]*"),
        ("OP", r"!=|<=|>=|[=<>,;()*]"), ("ESPACE", r"[ \t\r]+"), ("NL", r"\n"), ("
   ERR", r".")]
MOTIF = re.compile("|".join(f"(?P<{n}>{p})" for n, p in SPEC))

def tokenize_sql(source):
    ligne, debut, jetons = 1, 0, []
    for m in MOTIF.finditer(source):
        genre, texte, col = m.lastgroup, m.group(), m.start() - debut + 1
        if genre == "NL":
            ligne, debut = ligne + 1, m.end()
        elif genre == "ESPACE":
            continue
        elif genre == "CHAINE_OUVERTE":
            raise SyntaxError(f"chaîne non terminée ligne {ligne}, colonne {col}")
```

Pistes de correction (suite)

```
        elif genre == "ERR":
            raise SyntaxError(f"caractère inattendu {texte!r} ligne {ligne}, colon
   ne {col}")
        else:
            if genre == "ID" and texte.upper() in MOTS:
                genre, texte = "MOT", texte.upper()
            jetons.append((genre, texte, ligne, col))
    return jetons

jetons = tokenize_sql("select nom, age FROM membres\nWhere age >= 18 and ville = '
   Lyon';")
assert [j[1] for j in jetons][:3] == ["SELECT", "nom", ","]
assert jetons[0][:2] == ("MOT", "SELECT") and jetons[6][2] == 2
for sql in ("SELECT * FROM t WHERE a != 3;", "SELECT a FROM t WHERE b <= 2.5 OR c
   = 'x y';"):
    tokenize_sql(sql)
for mauvais in ("SELECT 'abc FROM t;", "SELECT a FROM t WHERE a @ 3;"):
    try:
        tokenize_sql(mauvais)
```

À toi, en solo · réponse

Pistes de correction (suite)

```
        raise AssertionError("aurait dû échouer")
    except SyntaxError as e:
        print(e)
```

Normaliser les mots-clés en majuscules évite au parser de se soucier de la casse.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Couper sur les espaces	x=3 n'est pas découpé	Reconnaître des motifs, pas des séparateurs
Mauvais ordre des motifs	= avant == coupe un opérateur en deux	Plus longs préfixes d'abord
Oublier les positions	Messages d'erreur inutiles	Suivre ligne et colonne dans le lexer
Ne pas gérer les caractères inconnus	Le lexer ignore silencieusement une erreur	Motif de repli ERR qui lève une erreur

Mini-boss

Mini-boss

Lexer robuste de HashScript. Écris un lexer complet qui reconnaît :

- nombres entiers et à virgule, identifiants, mots-clés, opérateurs (`==`, `!=`, `<=`, `>=`, `+`, `-`, `*`, `/`, `**` pour la puissance, parenthèses, accolades, virgule, point-virgule) ;
- chaînes entre guillemets avec échappements `\n`, `\"` et `\\` (la valeur du jeton doit contenir la chaîne décodée) ;
- commentaires `#…` et commentaires de bloc `/* … */` (qui peuvent s'étendre sur plusieurs lignes) ;
- suivi exact de ligne/colonne, y compris après un commentaire de bloc ;
- erreurs : caractère inconnu, chaîne ou commentaire de bloc non terminés (avec la position du **début**).

Écris au moins 20 tests, dont un test aléatoire : génère des programmes à partir de jetons, remets-les en texte avec des espaces aléatoires, et vérifie que le lexer retrouve les mêmes jetons.

Mini-boss · réponse

Pistes de correction

Un bon travail : les motifs `CHAINE` avec `"(?:[^"\\\n]|\\.)*"` pour gérer les échappements puis un décodage par `bytes(...).decode("unicode_escape")` ou une table de remplacement; le commentaire de bloc `/\*.*?\*/` avec le drapeau `re.S`; le calcul de ligne/colonne qui compte les `\n` à l'intérieur d'un jeton multi-lignes (`texte.count("\n")` et `texte.rfind("\n")` pour mettre à jour `debut_ligne`); les erreurs de chaîne/commentaire non terminés utilisent des motifs « ouverts » placés après les motifs fermés; le test aléatoire compare `[(genre, texte)]` avant et après insertion d'espaces et de commentaires entre les jetons. L'opérateur `**` est placé avant `*` dans la liste des motifs.

Quiz éclair

Quiz éclair

1. Un jeton est :

A. une instruction du processeur
B. un type de variable
C. une unité lexicale avec un genre, un texte et une position

2. Pourquoi reconnaître `<=` avant `<` ?

A. pour ne pas couper l'opérateur en deux
B. pour aller plus vite
C. par convention esthétique

3. Le rôle d'un lexer est de :

A. vérifier les types

Quiz éclair

Quiz éclair (suite)

B. transformer le texte en suite de jetons
C. exécuter le programme

4. 🔁 *Révision mission 01* : Une grammaire est ambiguë si :

A. elle est trop longue
B. elle utilise des regex
C. une phrase a plusieurs arbres syntaxiques

5. 🔁 *Révision mission 01* : Une expression régulière suffit pour :

A. reconnaître des mots (nombres, identifiants)
B. reconnaître des parenthèses équilibrées
C. exécuter un programme

Quiz éclair · réponse

Réponses

1. **C**.
2. **A**.
3. **B**.
4. **C**.
5. **A**.

Je réfléchis

1. Dans quelle situation de ta vie « découpes-tu » un texte en morceaux significatifs avant de le comprendre ?
2. Pourquoi mettre les positions dans les jetons dès le début est-il un investissement rentable ?

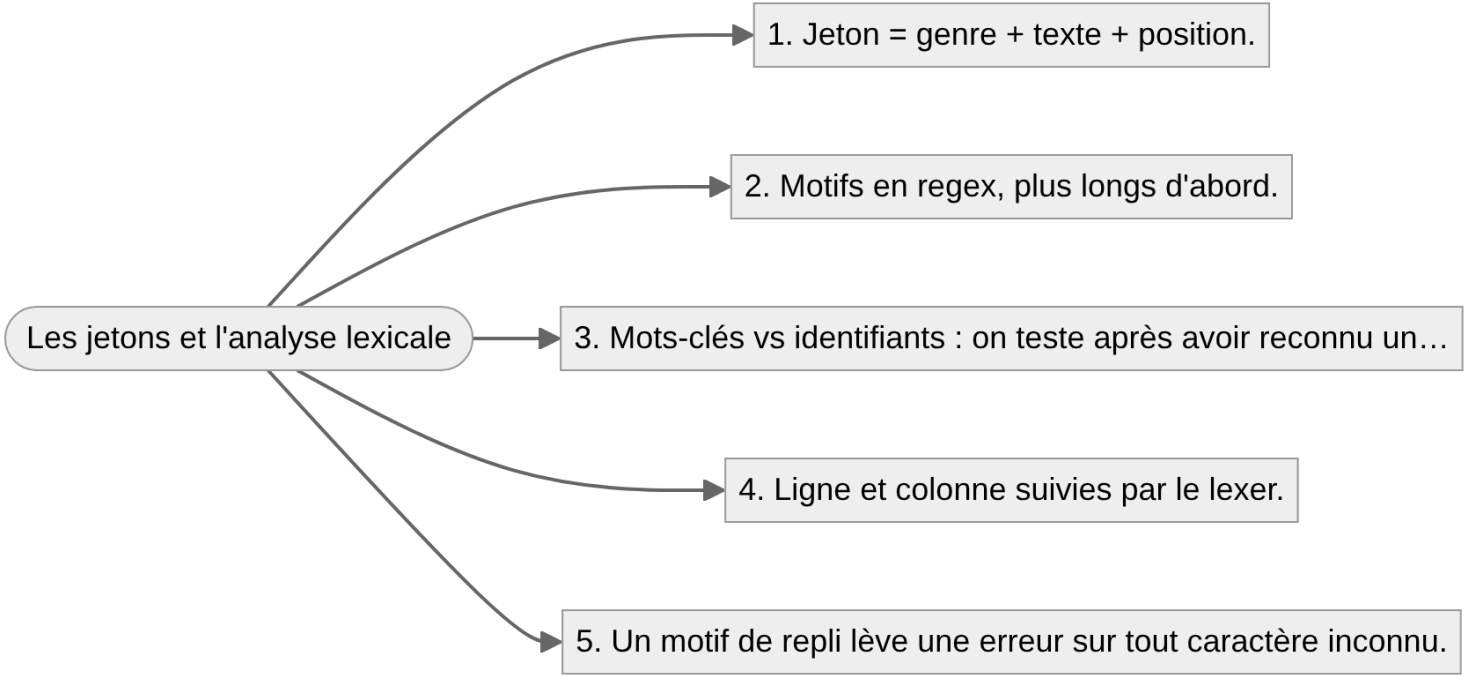
À retenir

À retenir

1. Jeton = genre + texte + position.
2. Motifs en regex, plus longs d'abord.
3. Mots-clés vs identifiants : on teste après avoir reconnu un identifiant.
4. Ligne et colonne suivies par le lexer.
5. Un motif de repli lève une erreur sur tout caractère inconnu.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Découpeur de mots

Prochaine mission : Le parsing par descente récursive

