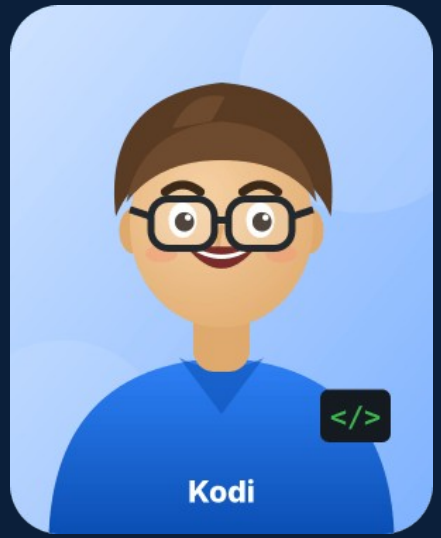
HEF-090 · MISSION 01 / 10

Langages et grammaires

Un langage de programmation est un langage comme les autres : il a un vocabulaire, une grammaire et un sens.



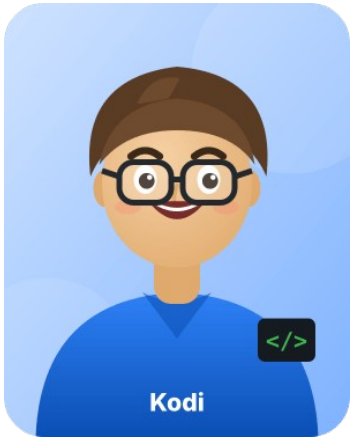


Badge : Linguiste des machines

80 min · intermédiaire

L'histoire

Maya demande à l'équipe de HashBox de créer un petit **langage de règles** pour que les formateurs puissent écrire eux-mêmes des conditions comme `si score > 50 alors badge`. Kodi pense : « Il suffit de chercher des mots-clés dans le texte. » Sam sourit :



> *« Essaie d'analyser si (score > 50) et (age < 18 ou accompagné) alors badge avec des recherches de mots. Pour comprendre une phrase, il faut une grammaire. Et pour la machine, c'est pareil. »*

Dans ce module, tu vas construire un vrai langage de A à Z : HashScript. Cette première mission pose les fondations : ce qu'est un langage, comment le décrire, comment la machine le comprend.

Ta mission

À la fin, je serai capable de…

1. distinguer syntaxe, sémantique et lexique
2. lire et écrire une grammaire simple en notation BNF
3. construire une dérivation et un arbre syntaxique
4. expliquer ce qu'est une grammaire ambiguë
5. décrire le chemin source → exécution (compilateur, interpréteur, machine virtuelle)

Défi d'échauffement

Défi d'échauffement

Voici deux phrases : « Le programme lit le fichier » et « Fichier le lit programme le ». La deuxième est-elle correcte ? Pourquoi ? Maintenant : « Le fichier dévore le silence ». La phrase est grammaticalement correcte, mais a-t-elle un **sens** ? Distingue ce qui relève (1) des mots utilisés, (2) de l'ordre des mots, (3) de la signification.

Le concept

Les trois niveaux d'un langage

Niveau	Question	Exemple (code)	Erreur typique
Lexical	quels « mots » (jetons) existent ?	`let`, `x`, `=`, `42`	caractère inconnu $
Syntaxique	comment les mots s'assemblent-ils ?	`let x = 42;`	`let = x 42`
Sémantique	que signifient les phrases ?	x doit être déclaré avant d'être lu	variable inconnue, types incompatibles

La machine comprend un programme en passant par ces niveaux dans cet ordre.

Le concept

Décrire un langage : la grammaire

Une **grammaire** est un ensemble de **règles de production** qui disent comment construire des phrases valides. On l'écrit en notation **BNF** (ou EBNF) :

```
phrase          → groupe_nominal verbe groupe_nominal
groupe_nominal  → article nom | article adjectif nom
article         → "le" | "un"
nom             → "compilateur" | "programme"
verbe           → "lit" | "traduit"
```

- Les mots entre guillemets sont des **terminaux** (les jetons réels).
- Les autres sont des **non-terminaux** (des catégories à développer).
- `|` signifie « ou ».

Le concept

Décrire un langage : la grammaire (suite)

Pour un langage de programmation :

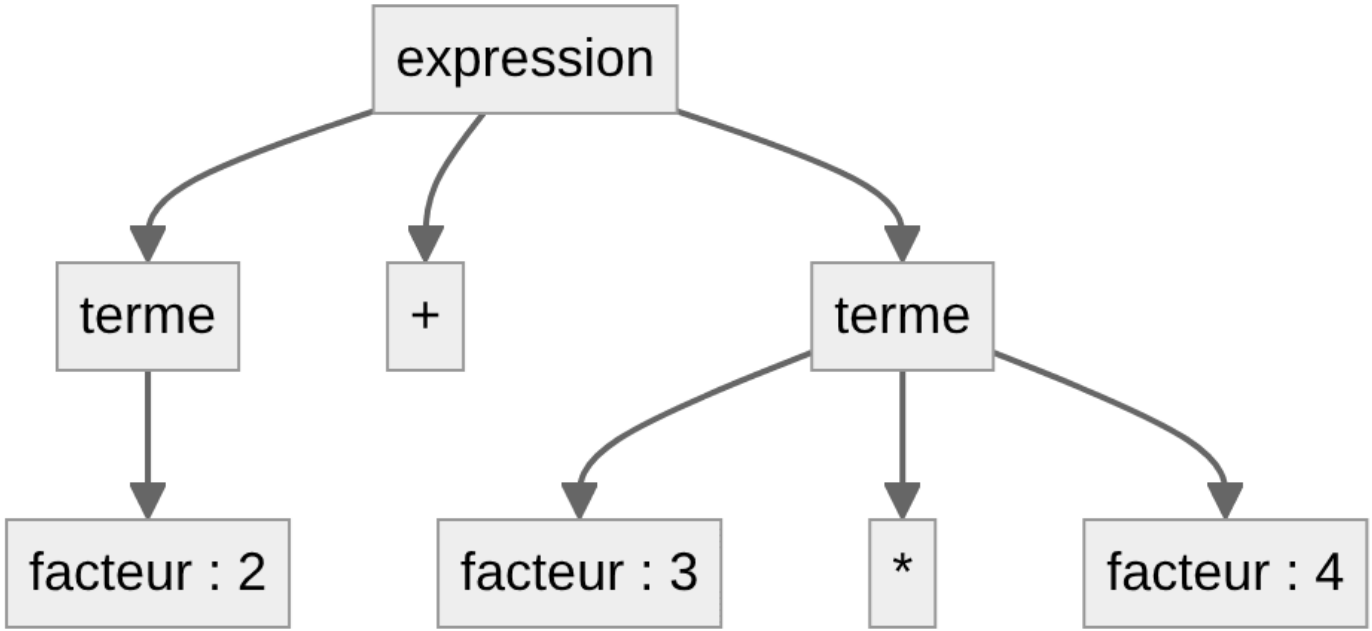
```
expression → terme (("+" | "-") terme)*
terme      → facteur (("*" | "/") facteur)*
facteur    → NOMBRE | IDENTIFIANT | "(" expression ")"
```

(EBNF : `( … )*` = zéro ou plusieurs répétitions.)

Dérivation et arbre syntaxique

Une **dérivation** part du symbole de départ et remplace des non-terminaux jusqu'à obtenir la phrase. On la représente par un **arbre syntaxique** (arbre de dérivation).

Dérivation et arbre syntaxique (suite)



Le concept

Dérivation et arbre syntaxique (suite)

Description du schéma : l'expression `2 + 3 * 4` se décompose en deux termes séparés par `+`. Le second terme contient lui-même `3 * 4`. La multiplication est donc « plus profonde » dans l'arbre : elle sera calculée avant l'addition. **La grammaire encode les priorités.**

L'ambiguïté

Une grammaire est **ambiguë** si une même phrase a plusieurs arbres. Exemple : `1 - 2 - 3` peut se lire `(1 - 2) - 3 = -4` ou `1 - (2 - 3) = 2`. Une grammaire de langage de programmation doit lever ces ambiguïtés (associativité à gauche, priorités). Autre classique : le « *else* pendant » (`if a if b x else y` : à quel `if` appartient le `else` ?).

Ce qu'une grammaire ne peut pas dire

Le concept

Ce qu'une grammaire ne peut pas dire (suite)

La grammaire décrit la **forme**, pas tout le sens. « Une variable doit être déclarée avant usage » ou « on ne peut pas additionner un booléen et du texte » sont des règles **sémantiques**, vérifiées dans une phase séparée (mission 5).

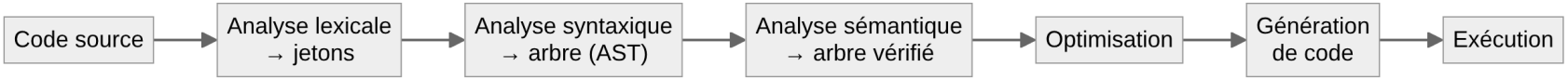
Expressions régulières ou grammaires ?

Outil	Capacité	Exemple
Expression régulière	motifs sans imbrication	reconnaître un nombre, un identifiant
Grammaire hors-contexte	structures **imbriquées**	parenthèses équilibrées, blocs `{ … }`

On ne peut pas reconnaître des parenthèses équilibrées avec une simple expression régulière : il faut « compter » ou « empiler ». C'est pourquoi on utilise des regex pour les mots (lexer) et des grammaires pour la structure (parser).

Le concept

Le chemin de la source à l'exécution



Description du schéma : le code source passe par l'analyse lexicale (jetons), l'analyse syntaxique (arbre), l'analyse sémantique (vérifications), l'optimisation, la génération de code, puis l'exécution.

Approche	Principe	Exemples
Compilateur	traduit tout le programme en code machine avant l'exécution	C, Rust, Go
Interpréteur	exécute directement l'arbre ou les instructions	premiers Python, Ruby
Machine virtuelle	compile vers un bytecode exécuté par une VM	Java, Python (CPython), C#

Le concept

Le chemin de la source à l'exécution (suite)

Approche	Principe	Exemples
JIT	compile à la volée les parties chaudes	JavaScript (V8), Java
Transpileur	traduit vers un autre langage de haut niveau	TypeScript → JavaScript

🧠 ***Astuce*** *: presque tout ce que tu écris contient un mini-langage (formats de configuration, règles, requêtes). Savoir construire un analyseur est une compétence utilisable partout.*

Exemple pas à pas

Problème : utiliser une grammaire pour **générer** des phrases, puis constater l'ambiguïté et la limite des regex.

```
import random

GRAMMAIRE = {
    "phrase": [["groupe_nominal", "verbe", "groupe_nominal"]],
    "groupe_nominal": [["article", "nom"], ["article", "adjectif", "nom"]],
    "article": [["le"], ["un"]],
    "adjectif": [["petit"], ["rapide"], ["curieux"]],
    "nom": [["compilateur"], ["programme"], ["lexer"]],
    "verbe": [["lit"], ["traduit"], ["exécute"]],
}

def generer(symbole, rng):
    if symbole not in GRAMMAIRE:                # terminal : un mot
        return [symbole]
    choix = rng.choice(GRAMMAIRE[symbole])      # on choisit une règle
```

Exemple pas à pas (suite)

```
    mots = []
    for s in choix:
        mots += generer(s, rng)
    return mots

rng = random.Random(3)
for _ in range(3):
    print(" ".join(generer("phrase", rng)))
```

Lecture : la grammaire engendre des phrases toujours **bien formées** (mais pas forcément sensées : « un petit lexer exécute le programme »). Le même principe permet de générer des programmes de test valides pour un compilateur (*fuzzing* guidé par grammaire).

Ambiguïté :

Exemple pas à pas

Exemple pas à pas (suite)

```
gauche = (1 - 2) - 3        # lecture "associative à gauche"
droite = 1 - (2 - 3)        # lecture "associative à droite"
print(gauche, droite)       # -4 2 : deux sens pour la même phrase
```

Limite des expressions régulières : reconnaître des parenthèses équilibrées demande un compteur (ou une pile).

```
def equilibre(texte):
    profondeur = 0
    for c in texte:
        if c == "(":
            profondeur += 1
        elif c == ")":
            profondeur -= 1
            if profondeur < 0:
                return False            # une fermante sans ouvrante
    return profondeur == 0
```

Exemple pas à pas (suite)

```
print(equilibre("((a+b)*(c))"), equilibre("(()"), equilibre(")("))   # True False
   False
```

Aucun motif de regex (sans extensions non standard) ne peut faire ce travail pour une profondeur arbitraire : il faut de la mémoire (un compteur ou une pile). C'est la différence entre langages **réguliers** et **hors-contexte**.

À toi, avec aide

À toi, avec aide

Avec la grammaire d'expressions arithmétiques :

```
expression → terme (("+" | "-") terme)*
terme      → facteur (("*" | "/") facteur)*
facteur    → NOMBRE | "(" expression ")"
```

1. Dessine (en texte indenté) l'arbre syntaxique de `2 * (3 + 4)`.
2. Dis dans quel ordre sont calculées les opérations d'après l'arbre.
3. Explique pourquoi cette grammaire n'est **pas** ambiguë pour `1 - 2 - 3`.

À toi, avec aide · réponse

Indices progressifs

Indice 1

`expression` commence par un `terme` ; ici ce terme est `2 * (3 + 4)`.

Indice 2

Le facteur `( … )` contient lui-même une `expression`.

Indice 3

La boucle `(…)*` de la règle impose de regrouper de gauche à droite.

À toi, avec aide · réponse

Réponse

```
expression
└── terme
    ├── facteur : 2
    ├── "*"
    └── facteur
        ├── "("
        ├── expression
        │   ├── terme → facteur : 3
        │   ├── "+"
        │   └── terme → facteur : 4
        └── ")"
```

1. L'arbre ci-dessus.
2. Les feuilles les plus profondes sont calculées en premier : `3 + 4` = 7, puis `2 * 7` = 14.
3. La règle `terme (("+" | "-") terme)*` fait un seul choix de regroupement : on lit les termes de gauche à droite, donc `(1 - 2) - 3`. Il n'existe qu'un arbre pour cette phrase.

À toi, en solo

Reconnaisseur de listes imbriquées. Écris la grammaire (dans un commentaire) de listes de nombres imbriquées comme `[1, [2, 3], 4]` :

```
liste   → "[" (element ("," element)*)? "]"
element → NOMBRE | liste
```

puis un reconnaisseur récursif `reconnait(texte)` qui répond `True` ou `False` (sans regex pour la structure). Teste au moins 10 chaînes : valides (`[]`, `[1]`, `[1, [2, [3]]]`) et invalides (`[1, 2`, `[1,,2]`, `[,]`, `[1 2]`, `]`).

À toi, en solo · réponse

Pistes de correction

```
def reconnait(texte):
    i = 0

    def espaces():
        nonlocal i
        while i < len(texte) and texte[i] == " ":
            i += 1

    def nombre():
        nonlocal i
        j = i
        while j < len(texte) and texte[j].isdigit():
            j += 1
        if j == i:
            return False
        i = j
        return True

    def element():
```

À toi, en solo · réponse

Pistes de correction (suite)

```
        espaces()
        if i < len(texte) and texte[i] == "[":
            return liste()
        return nombre()

    def liste():
        nonlocal i
        i += 1                                          # "["
        espaces()
        if i < len(texte) and texte[i] == "]":
            i += 1
            return True
        if not element():
            return False
        espaces()
        while i < len(texte) and texte[i] == ",":
            i += 1
            if not element():
                return False
```

Pistes de correction (suite)

```
            espaces()
        if i < len(texte) and texte[i] == "]":
            i += 1
            return True
        return False

    if not texte.strip().startswith("["):
        return False
    ok = liste() if texte[0] == "[" else (espaces() or liste())
    espaces()
    return ok and i == len(texte)

valides = ["[]", "[1]", "[1, 2, 3]", "[1, [2, [3]]]", "[[], []]", " [ 1 , 2 ] "]
invalides = ["[1, 2", "[1,,2]", "[,]", "[1 2]", "]", "[1, [2]", "", "[1] 2"]
assert all(reconnait(t) for t in valides), [t for t in valides if not reconnait(t)
   ]
assert not any(reconnait(t) for t in invalides), [t for t in invalides if reconnai
   t(t)]
print("grammaire reconnue")
```

Pistes de correction (suite)

La structure récursive du reconnaisseur **épouse** la grammaire : une fonction par non-terminal. C'est exactement le principe de l'analyse par descente récursive (mission 3).

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Confondre syntaxe et sémantique	On cherche une erreur de type dans la grammaire	Forme d'abord, sens ensuite
Grammaire ambiguë	Plusieurs sens pour la même phrase	Encoder priorités et associativité dans les règles
Tout faire avec des regex	Impossible pour les structures imbriquées	Regex pour les mots, grammaire pour la structure
Mélanger les phases	Code illisible, erreurs mal localisées	Lexer, parser, sémantique : phases séparées

Mini-boss

Mini-boss

Grammaire d'un petit langage de règles. Les formateurs de HashBox veulent écrire des règles comme :

```
si score >= 50 et (age < 18 ou accompagne) alors badge "bronze"
```

1. Écris la grammaire EBNF complète : règles, conditions avec `et` / `ou` / `non`, comparaisons, parenthèses, actions (`badge "texte"`, `message "texte"`).
2. Fixe les priorités (`non` > `et` > `ou`) dans la grammaire et explique pourquoi elle n'est pas ambiguë.
3. Écris un **générateur** aléatoire de règles valides (graine fixée, profondeur limitée) et produis 10 règles.
4. Écris un **reconnaisseur** simple (par descente récursive sur une liste de jetons obtenue avec `str.split` ou une regex) qui valide les règles générées et rejette 5 règles volontairement invalides.

Mini-boss · réponse

Pistes de correction

Grammaire : `regle → "si" condition "alors" action` ; `condition → ou` ; `ou → et ("ou" et)*` ; `et → non ("et" non)*` ; `non → "non" non | atome` ; `atome → comparaison | "(" condition ")" | IDENT` ; `comparaison → IDENT OP NOMBRE` ; `action → ("badge" | "message") CHAINE`. Les priorités viennent de l'imbrication des règles : `ou` contient des `et`, qui contiennent des `non`. Le générateur choisit récursivement une règle, avec une profondeur maximale pour s'arrêter. Le reconnaisseur utilise une fonction par non-terminal avec un indice courant sur la liste de jetons. Les cinq invalides : parenthèse manquante, `et` sans second opérande, action inconnue, opérateur de comparaison absent, mot-clé `alors` oublié.

Quiz éclair

Quiz éclair

1. La syntaxe d'un langage décrit :

A. comment les jetons s'assemblent
B. la signification des programmes
C. la vitesse d'exécution

2. Un terminal d'une grammaire est :

A. une catégorie à développer
B. un symbole réel du langage (jeton)
C. un type de compilateur

3. Une grammaire est ambiguë si :

A. elle est trop longue

Quiz éclair

Quiz éclair (suite)

B. elle utilise des regex
C. une phrase a plusieurs arbres syntaxiques

4. 🔁 *Révision HEF-070, mission 08* : Une transaction garantit :

A. tout ou rien
B. des requêtes plus rapides
C. des données chiffrées

5. 🔁 *Révision HEF-080, mission 10* : Un ADR sert à :

A. lister les bugs
B. garder une décision et ses raisons
C. remplacer les tests

Réponses

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

Je réfléchis

1. Quels « mini-langages » utilises-tu déjà sans les nommer (formules de tableur, filtres de recherche, expressions régulières) ?
2. Pourquoi séparer lexique, syntaxe et sémantique simplifie-t-il la construction d'un langage ?

À retenir

À retenir

1. Trois niveaux : lexical, syntaxique, sémantique.
2. Grammaire BNF : terminaux, non-terminaux, règles.
3. Dérivation et arbre syntaxique encodent les priorités.
4. Ambiguïté : plusieurs arbres pour une phrase.
5. Pipeline : lexer → parser → sémantique → optimisation → génération/exécution.

MISSION ACCOMPLIE

+100 XP

Badge débloqué : Linguiste des machines

Prochaine mission : Les jetons et l'analyse lexicale

