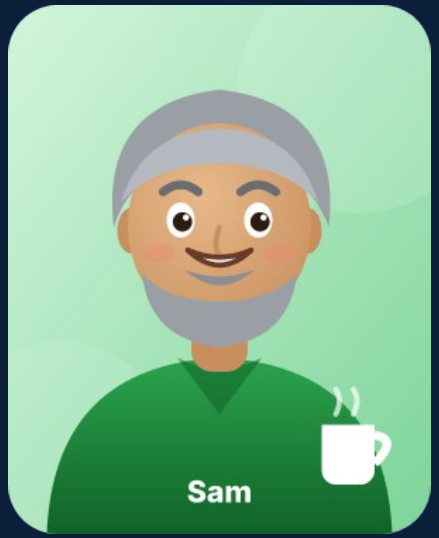
HEF-090 · MISSION 03 / 10

Le parsing par descente récursive

Une fonction par règle de grammaire : la grammaire devient du code, et l'arbre apparaît tout seul.





Badge : Grammairien du code

90 min · intermédiaire

L'histoire

Le lexer de Kodi produit des jetons. Maintenant il faut comprendre `2 + 3 * 4`. Kodi essaie avec une boucle qui lit de gauche à droite : résultat 20 au lieu de 14.



> *« Tu ignores la grammaire, dit Sam. La structure du texte est un arbre. Chaque règle de grammaire devient une fonction, et les priorités sortent naturellement de l'imbrication des appels. »*

Tu écris le parser de HashScript par descente récursive.

Ta mission

À la fin, je serai capable de…

1. transformer une grammaire en fonctions récursives
2. encoder priorités et associativité dans la structure des règles
3. construire un arbre syntaxique pendant l'analyse
4. signaler les erreurs de syntaxe avec un message utile
5. évaluer un arbre d'expression

Défi d'échauffement

Calcule à la main `2 + 3 * 4 - 1`. Dans quel ordre as-tu fait les opérations ? Dessine l'arbre des calculs (un nœud par opération). Maintenant, suppose que tu veuilles écrire un programme qui lit `2 + 3 * 4 - 1` de gauche à droite : à quel moment sait-il qu'il doit faire la multiplication avant l'addition ?

Le concept

L'idée

Une **grammaire** devient un **programme** : on écrit **une fonction par non-terminal**. Chaque fonction lit les jetons qui correspondent à sa règle et appelle d'autres fonctions pour les sous-règles. C'est l'**analyse par descente récursive** (*recursive descent*).

Grammaire des expressions (EBNF) :

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

Priorités par l'imbrication

Le concept

Priorités par l'imbrication (suite)

`expression` appelle `terme`, qui appelle `facteur`. Les opérateurs les plus **prioritaires** (`*`, `/`) sont donc traités **plus profondément** dans l'arbre, donc calculés en premier. Ajouter un niveau de priorité = ajouter une règle (et une fonction) entre deux niveaux.

Associativité par la boucle

La boucle `( … )*` construit l'arbre **vers la gauche** :

```
1 - 2 - 3 :  g = 1
             g = (1 - 2)
             g = ((1 - 2) - 3)
```

C'est l'associativité à gauche. Pour un opérateur associatif à droite (comme la puissance `^`), la règle s'écrit récursivement : `puissance → facteur ("^" puissance)?`.

Le concept

Le parser en Python

```
import re

def lexer(texte):
    return re.findall(r"\d+(?:\.\d+)?|[-+*/()]", texte)

class Parseur:
    def __init__(self, jetons):
        self.jetons, self.i = jetons, 0

    def courant(self):
        return self.jetons[self.i] if self.i < len(self.jetons) else None

    def prendre(self):
        jeton = self.courant()
        self.i += 1
        return jeton
```

Le concept

Le parser en Python (suite)

```
    def expression(self):                       # expression → terme (("+" | "-")
terme)*
        gauche = self.terme()
        while self.courant() in ("+", "-"):
            op = self.prendre()
            gauche = ("bin", op, gauche, self.terme())
        return gauche

    def terme(self):                            # terme → facteur (("*" | "/") fac
teur)*
        gauche = self.facteur()
        while self.courant() in ("*", "/"):
            op = self.prendre()
            gauche = ("bin", op, gauche, self.facteur())
        return gauche

    def facteur(self):                          # facteur → NOMBRE | "(" expressio
n ")" | "-" facteur
```

Le concept

Le parser en Python (suite)

```
        jeton = self.prendre()
        if jeton is None:
            raise SyntaxError("expression incomplète")
        if jeton == "(":
            e = self.expression()
            if self.prendre() != ")":
                raise SyntaxError("parenthèse fermante attendue")
            return e
        if jeton == "-":
            return ("neg", self.facteur())
        if re.fullmatch(r"\d+(?:\.\d+)?", jeton):
            return ("num", float(jeton) if "." in jeton else int(jeton))
        raise SyntaxError(f"jeton inattendu : {jeton}")

def analyser(texte):
    p = Parseur(lexer(texte))
    arbre = p.expression()
    if p.courant() is not None:
        raise SyntaxError(f"jeton inattendu : {p.courant()}")
```

Le parser en Python (suite)

```
    return arbre
```

Évaluer l'arbre

```
OPS = {"+": lambda x, y: x + y, "-": lambda x, y: x - y, "*": lambda x, y: x * y,
   "/": lambda x, y: x / y}

def evaluer(a):
    if a[0] == "num":
        return a[1]
    if a[0] == "neg":
        return -evaluer(a[1])
    _, op, g, d = a
    return OPS[op](evaluer(g), evaluer(d))
```

Le concept

Les erreurs de syntaxe

Un bon parser dit **ce qu'il attendait** et **ce qu'il a trouvé** : « attendu `)`, trouvé fin du programme ». Il vérifie aussi qu'il n'y a **rien après** l'expression complète.

Limites de la descente récursive

- Elle ne supporte pas la **récursion à gauche** (`expr → expr "+" terme` bouclerait indéfiniment) : on la transforme en boucle.
- Pour de grandes grammaires, on utilise des **générateurs de parseurs** (ANTLR, Lark, PEG) ou des analyseurs LR (Yacc/Bison). Les principes restent les mêmes.

🧠 ***Astuce** : écris d'abord la grammaire sur papier. Chaque ligne de grammaire devient une fonction. Si le code ne ressemble pas à la grammaire, quelque chose ne va pas.*

Exemple pas à pas

Problème : analyser et évaluer des expressions, en vérifiant les priorités et les erreurs.

```
import re

def lexer(texte):
    return re.findall(r"\d+(?:\.\d+)?|[-+*/()]", texte)

class Parseur:
    def __init__(self, jetons):
        self.jetons, self.i = jetons, 0
    def courant(self):
        return self.jetons[self.i] if self.i < len(self.jetons) else None
    def prendre(self):
        jeton = self.courant(); self.i += 1; return jeton
    def expression(self):
        gauche = self.terme()
        while self.courant() in ("+", "-"):
            op = self.prendre(); gauche = ("bin", op, gauche, self.terme())
```

Exemple pas à pas (suite)

```
        return gauche
    def terme(self):
        gauche = self.facteur()
        while self.courant() in ("*", "/"):
            op = self.prendre(); gauche = ("bin", op, gauche, self.facteur())
        return gauche
    def facteur(self):
        jeton = self.prendre()
        if jeton is None:
            raise SyntaxError("expression incomplète")
        if jeton == "(":
            e = self.expression()
            if self.prendre() != ")":
                raise SyntaxError("parenthèse fermante attendue")
            return e
        if jeton == "-":
            return ("neg", self.facteur())
        if re.fullmatch(r"\d+(?:\.\d+)?", jeton):
            return ("num", float(jeton) if "." in jeton else int(jeton))
```

Exemple pas à pas (suite)

```
        raise SyntaxError(f"jeton inattendu : {jeton}")

def analyser(texte):
    p = Parseur(lexer(texte))
    arbre = p.expression()
    if p.courant() is not None:
        raise SyntaxError(f"jeton inattendu : {p.courant()}")
    return arbre

OPS = {"+": lambda x, y: x + y, "-": lambda x, y: x - y, "*": lambda x, y: x * y,
   "/": lambda x, y: x / y}

def evaluer(a):
    if a[0] == "num": return a[1]
    if a[0] == "neg": return -evaluer(a[1])
    _, op, g, d = a
    return OPS[op](evaluer(g), evaluer(d))

print(analyser("2 + 3 * 4"))
```

Exemple pas à pas (suite)

```
# ('bin', '+', ('num', 2), ('bin', '*', ('num', 3), ('num', 4)))
for texte in ("2 + 3 * 4", "(2 + 3) * 4", "10 - 4 - 3", "-(1 + 2) * 3", "2 * -3"):
    attendu = eval(texte)                                  # on compare à Python (rés
   ervé aux tests !)
    assert evaluer(analyser(texte)) == attendu, texte
    print(f"{texte:14s} = {evaluer(analyser(texte))}")

for mauvais in ("2 +", "(1 + 2", "1 2", "* 3"):
    try:
        analyser(mauvais)
    except SyntaxError as e:
        print(f"{mauvais!r:10} → {e}")
```

Lecture :

- l'arbre de `2 + 3 * 4` place `3 * 4` **sous** l'addition : le résultat est 14, pas 20 ;
- `10 - 4 - 3` donne 3 (associativité à gauche) et non 9 ;

Exemple pas à pas (suite)

- on compare chaque résultat à `eval` de Python **uniquement dans les tests** pour vérifier le parser (jamais `eval` sur une entrée utilisateur : danger !) ;
- les quatre entrées invalides produisent chacune un message précis.

À toi, avec aide

Ajoute au parseur de l'exemple les **comparaisons** : `<`, `>`, `<=`, `>=`, `==`, `!=`, avec une priorité **plus basse** que l'addition (donc `1 + 2 < 4` se lit `(1 + 2) < 4`) et un résultat booléen.

1. Écris la nouvelle grammaire.
2. Modifie le lexer (les opérateurs à deux caractères) et le parseur.
3. Étends `evaluer`.
4. Teste : `1 + 2 < 4` → True, `2 * 3 == 6` → True, `5 != 5` → False.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Nouvelle règle : `comparaison → expression (OP_COMP expression)?` placée au-dessus de `expression`.

Indice 2

Le lexer doit reconnaître `<=|>=|==|!=|[<>]` avant les opérateurs simples.

Indice 3

Appelle `comparaison` depuis `analyser` à la place de `expression`, et depuis les parenthèses dans `facteur`.

À toi, avec aide · réponse

Réponse

```
import re

def lexer(texte):
    return re.findall(r"\d+(?:\.\d+)?|<=|>=|==|!=|[<>]|[-+*/()]", texte)

class Parseur:
    def __init__(self, jetons):
        self.jetons, self.i = jetons, 0
    def courant(self):
        return self.jetons[self.i] if self.i < len(self.jetons) else None
    def prendre(self):
        jeton = self.courant(); self.i += 1; return jeton
    def comparaison(self):                          # comparaison → expression (O
   P_COMP expression)?
        gauche = self.expression()
        if self.courant() in ("<", ">", "<=", ">=", "==", "!="):
            op = self.prendre()
            return ("cmp", op, gauche, self.expression())
        return gauche
```

À toi, avec aide · réponse

Réponse (suite)

```
    def expression(self):
        gauche = self.terme()
        while self.courant() in ("+", "-"):
            op = self.prendre(); gauche = ("bin", op, gauche, self.terme())
        return gauche
    def terme(self):
        gauche = self.facteur()
        while self.courant() in ("*", "/"):
            op = self.prendre(); gauche = ("bin", op, gauche, self.facteur())
        return gauche
    def facteur(self):
        jeton = self.prendre()
        if jeton is None:
            raise SyntaxError("expression incomplète")
        if jeton == "(":
            e = self.comparaison()
            if self.prendre() != ")":
                raise SyntaxError("parenthèse fermante attendue")
            return e
```

Réponse (suite)

```
        if jeton == "-":
            return ("neg", self.facteur())
        if re.fullmatch(r"\d+(?:\.\d+)?", jeton):
            return ("num", float(jeton) if "." in jeton else int(jeton))
        raise SyntaxError(f"jeton inattendu : {jeton}")

def analyser(texte):
    p = Parseur(lexer(texte))
    arbre = p.comparaison()
    if p.courant() is not None:
        raise SyntaxError(f"jeton inattendu : {p.courant()}")
    return arbre

OPS = {"+": lambda x, y: x + y, "-": lambda x, y: x - y, "*": lambda x, y: x * y,
   "/": lambda x, y: x / y,
       "<": lambda x, y: x < y, ">": lambda x, y: x > y, "<=": lambda x, y: x <= y
   , ">=": lambda x, y: x >= y,
       "==": lambda x, y: x == y, "!=": lambda x, y: x != y}
```

À toi, avec aide · réponse

Réponse (suite)

```
def evaluer(a):
    if a[0] == "num": return a[1]
    if a[0] == "neg": return -evaluer(a[1])
    _, op, g, d = a
    return OPS[op](evaluer(g), evaluer(d))

assert evaluer(analyser("1 + 2 < 4")) is True
assert evaluer(analyser("2 * 3 == 6")) is True
assert evaluer(analyser("5 != 5")) is False
assert evaluer(analyser("(1 < 2) == (3 > 2)")) is True
print("comparaisons ok")
```

À toi, en solo

La puissance. Ajoute l'opérateur `^` (puissance), qui doit être :

- **plus prioritaire** que `*` et `/` ;
- **associatif à droite** : `2 ^ 3 ^ 2` = `2 ^ (3 ^ 2)` = 512 ;
- plus prioritaire que le moins unaire dans la convention mathématique : `-2 ^ 2` = `-(2 ^ 2)` = -4.

Écris la nouvelle grammaire, modifie lexer et parseur, et teste : `2 ^ 3 ^ 2`, `-2 ^ 2`, `(2 ^ 3) ^ 2`, `2 * 3 ^ 2`, `2 ^ -1`.

À toi, en solo · réponse

Pistes de correction

```
terme      → unaire (("*" | "/") unaire)*
unaire     → "-" unaire | puissance
puissance  → atome ("^" unaire)?           # associatif à droite, l'exposant peut
    être négatif
atome      → NOMBRE | "(" expression ")"
```

```
import re

def lexer(texte):
    return re.findall(r"\d+(?:\.\d+)?|[-+*/^()]", texte)

class Parseur:
    def __init__(self, jetons):
        self.jetons, self.i = jetons, 0
    def courant(self):
        return self.jetons[self.i] if self.i < len(self.jetons) else None
    def prendre(self):
        jeton = self.courant(); self.i += 1; return jeton
```

Pistes de correction (suite)

```
    def expression(self):
        g = self.terme()
        while self.courant() in ("+", "-"):
            op = self.prendre(); g = ("bin", op, g, self.terme())
        return g
    def terme(self):
        g = self.unaire()
        while self.courant() in ("*", "/"):
            op = self.prendre(); g = ("bin", op, g, self.unaire())
        return g
    def unaire(self):
        if self.courant() == "-":
            self.prendre()
            return ("neg", self.unaire())
        return self.puissance()
    def puissance(self):
        base = self.atome()
        if self.courant() == "^":
            self.prendre()
```

Pistes de correction (suite)

```
            return ("bin", "^", base, self.unaire())       # appel récursif à droi
   te : associativité à droite
        return base
    def atome(self):
        jeton = self.prendre()
        if jeton is None:
            raise SyntaxError("expression incomplète")
        if jeton == "(":
            e = self.expression()
            if self.prendre() != ")":
                raise SyntaxError("parenthèse fermante attendue")
            return e
        if re.fullmatch(r"\d+(?:\.\d+)?", jeton):
            return ("num", float(jeton) if "." in jeton else int(jeton))
        raise SyntaxError(f"jeton inattendu : {jeton}")

def analyser(texte):
    p = Parseur(lexer(texte))
    a = p.expression()
```

Pistes de correction (suite)

```
    if p.courant() is not None:
        raise SyntaxError(f"jeton inattendu : {p.courant()}")
    return a

OPS = {"+": lambda x, y: x + y, "-": lambda x, y: x - y, "*": lambda x, y: x * y,
   "/": lambda x, y: x / y, "^": lambda x, y: x ** y}

def evaluer(a):
    if a[0] == "num": return a[1]
    if a[0] == "neg": return -evaluer(a[1])
    _, op, g, d = a
    return OPS[op](evaluer(g), evaluer(d))

for texte, attendu in (("2 ^ 3 ^ 2", 512), ("-2 ^ 2", -4), ("(2 ^ 3) ^ 2", 64), ("
   2 * 3 ^ 2", 18), ("2 ^ -1", 0.5)):
    assert evaluer(analyser(texte)) == attendu, texte
print("puissance ok")
```

Pistes de correction (suite)

L'associativité à droite vient de l'appel récursif **à droite** (`self.unaire()` pour l'exposant) au lieu d'une boucle.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Lire de gauche à droite sans structure	`2 + 3 * 4` donne 20	Une fonction par niveau de priorité
Récursion à gauche	`expression → expression + terme` boucle à l'infini	La transformer en boucle `(...)*`
Oublier de vérifier la fin de l'entrée	`1 2` est accepté	Contrôler qu'il ne reste aucun jeton
Messages d'erreur vagues	L'utilisateur ne sait pas quoi corriger	Dire ce qui était attendu et ce qui a été trouvé

Mini-boss

Mini-boss

Parseur de HashScript (expressions et appels). Étends le parseur d'expressions pour obtenir la grammaire :

```
expression  → comparaison
comparaison → somme (OP_COMP somme)?
somme       → produit (("+" | "-") produit)*
produit     → unaire (("*" | "/") unaire)*
unaire      → "-" unaire | appel
appel       → primaire ("(" arguments? ")")*
primaire    → NOMBRE | IDENTIFIANT | "true" | "false" | "(" expression ")"
```

1. Le parseur doit produire des nœuds `("appel", fonction, [arguments])` et `("var", nom)`.
2. Écris `afficher(arbre)` qui imprime l'expression complètement parenthésée (pour vérifier les priorités).
3. Teste au moins 15 expressions, dont `f(1)(2)`, `-f(x) * 2`, `a < b == c`.

Mini-boss

Mini-boss (suite)

4. Les messages d'erreur doivent indiquer le jeton attendu et le jeton trouvé.
5. Teste aussi que 200 expressions aléatoires, générées par une fonction qui parenthèse complètement, sont analysées à l'identique de leur version avec parenthèses minimales.

Mini-boss · réponse

Pistes de correction

Un bon travail : `appel` boucle tant que `(` suit, ce qui permet `f(1)(2)` ; `afficher` produit `((a < b) == c)` pour montrer l'associativité de la comparaison ; `primaire` renvoie `("var", nom)` pour les identifiants et `("bool", True)` pour `true` ; les erreurs disent « attendu `)`, trouvé fin de l'entrée » ; le test aléatoire génère un arbre d'expressions, l'imprime avec parenthèses **complètes** puis avec parenthèses **minimales** (en utilisant les priorités), et vérifie que `afficher(analyser(texte_min)) == afficher(analyser(texte_complet))`.

Quiz éclair

Quiz éclair

1. Dans la descente récursive, on écrit :

A. un seul gros `if`
B. une fonction par règle de grammaire
C. une expression régulière

2. Les priorités entre opérateurs sont encodées par :

A. l'ordre des jetons dans le fichier
B. une table triée au hasard
C. l'imbrication des règles (qui appelle qui)

3. La boucle `( … )*` dans une règle produit :

A. une associativité à gauche

Quiz éclair

Quiz éclair (suite)

B. une associativité à droite
C. aucune associativité

4. 🔁 *Révision mission 01* : Un terminal d'une grammaire est :

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

5. 🔁 *Révision mission 02* : Un jeton est :

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

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Dans ta vie, où appliques-tu des priorités implicites (ordre des opérations, ordre des tâches) sans les écrire ?
2. Qu'est-ce qui rend la descente récursive attrayante pour un débutant par rapport à d'autres techniques ?

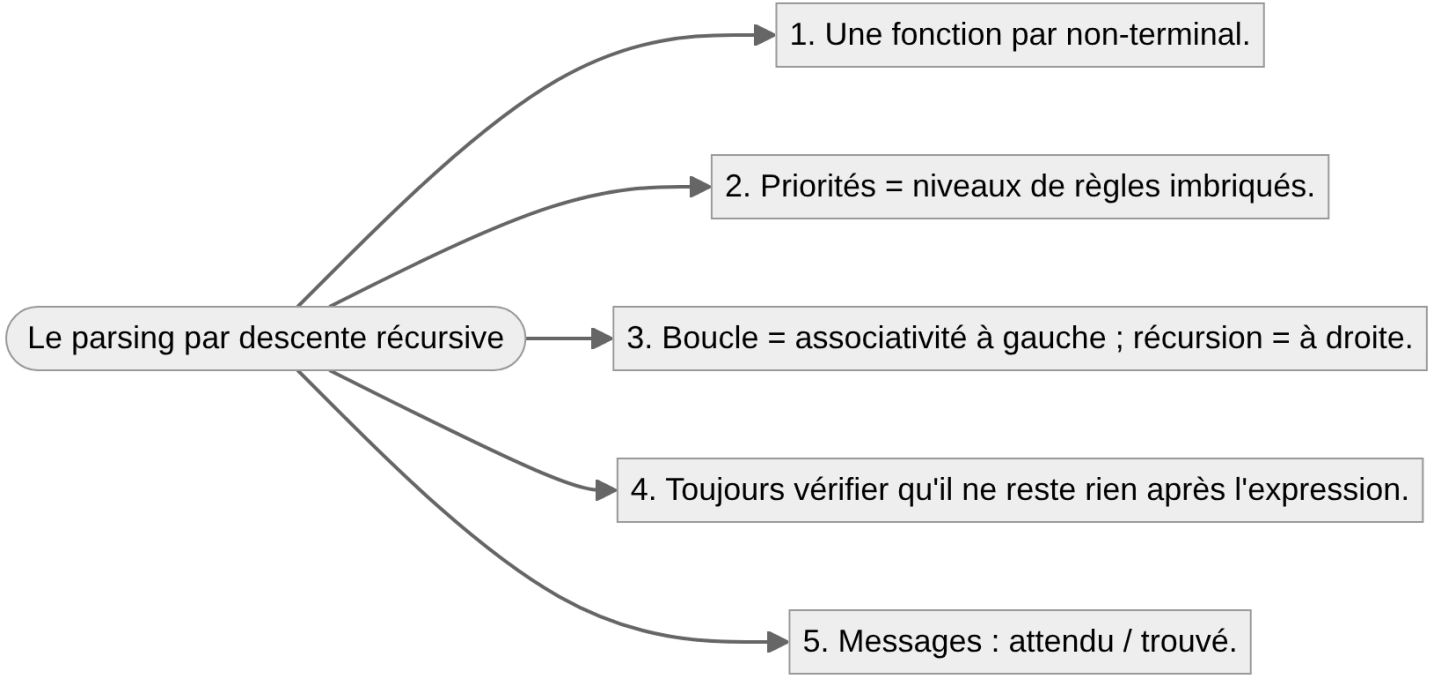
À retenir

À retenir

1. Une fonction par non-terminal.
2. Priorités = niveaux de règles imbriqués.
3. Boucle = associativité à gauche ; récursion = à droite.
4. Toujours vérifier qu'il ne reste rien après l'expression.
5. Messages : attendu / trouvé.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Grammairien du code

Prochaine mission : L'arbre syntaxique abstrait (AST)

