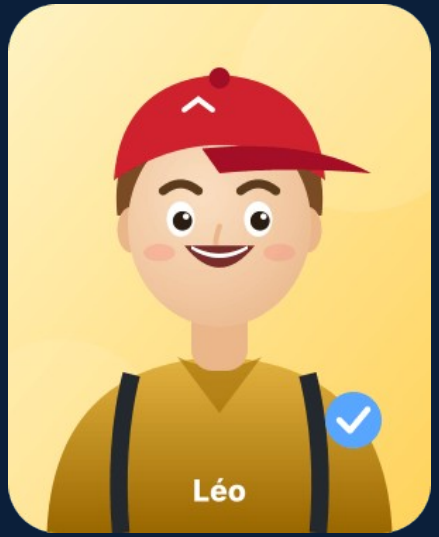
HEF-090 · MISSION 04 / 10

L'arbre syntaxique abstrait (AST)

Après le parsing, le programme n'est plus un texte : c'est un arbre qu'on peut parcourir, transformer et optimiser.



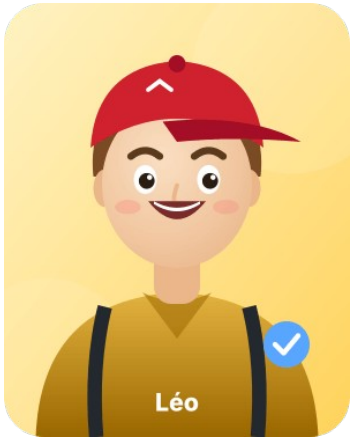


Badge : Jardinier d'arbres

85 min · intermédiaire

L'histoire

Kodi dispose maintenant d'un parser. Maya demande : « Peux-tu m'afficher la liste de toutes les variables utilisées dans un programme HashScript ? Et simplifier `2 * 3 + x` en `6 + x` automatiquement ? » Kodi cherche à le faire sur le texte et se perd dans les cas particuliers.



> *« Le texte est une représentation pour les humains, dit Sam. Pour les machines, l'arbre est la bonne structure : on le parcourt, on le transforme, on l'optimise. »*

Tu apprends à représenter, parcourir et transformer un AST.

Ta mission

À la fin, je serai capable de…

1. expliquer la différence entre arbre syntaxique concret et AST
2. lire et construire un AST avec des nœuds typés (dataclasses)
3. écrire un parcours récursif (visiteur) sur un AST
4. implémenter une transformation : le pliage de constantes
5. produire une représentation de l'arbre (S-expression, Mermaid)

Défi d'échauffement

Défi d'échauffement

Voici l'expression `(2 + 3) * x`. Dessine l'arbre qui la représente en gardant **seulement ce qui compte pour le calcul** (opérateurs, nombres, variables), sans les parenthèses ni les virgules. Que deviennent les parenthèses ? Peux-tu retrouver le texte à partir de l'arbre ?

Le concept

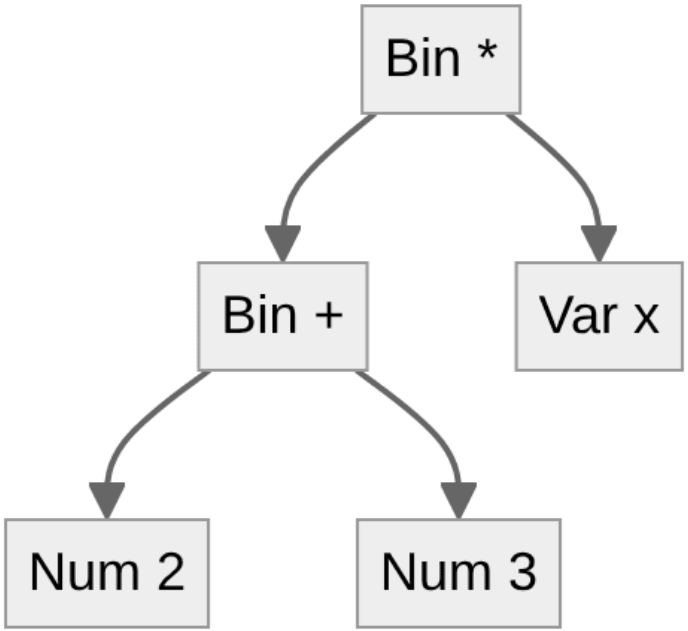
Arbre concret et AST

- L'**arbre syntaxique concret** (de dérivation) contient **tous** les détails de la grammaire (règles intermédiaires, parenthèses, séparateurs).
- L'**AST** (*Abstract Syntax Tree*) garde **l'essentiel** : la structure du programme, sans le « bruit » syntaxique.

Les parenthèses de `(2 + 3) * x` disparaissent de l'AST : leur effet est déjà dans la **forme** de l'arbre.

Le concept

Arbre concret et AST (suite)



Le concept

Arbre concret et AST (suite)

Description du schéma : la racine est une multiplication. Son enfant gauche est l'addition de 2 et 3 ; son enfant droit est la variable x.

Des nœuds typés

En Python, on représente un AST avec des classes (souvent des `@dataclass`) :

```
from dataclasses import dataclass

@dataclass
class Num: valeur: float
@dataclass
class Var: nom: str
@dataclass
class Bin: op: str; gauche: object; droite: object
@dataclass
class Neg: expr: object
```

Le concept

Des nœuds typés (suite)

Avantages par rapport à des tuples : noms de champs explicites, vérification des types, affichage lisible, positions de code (ligne, colonne) ajoutables.

Parcourir un arbre : le visiteur

Presque tout traitement est un **parcours récursif** : on regarde le **type** du nœud, on traite, on descend dans les enfants.

```
def afficher(n):
    if isinstance(n, Num):  return str(n.valeur)
    if isinstance(n, Var):  return n.nom
    if isinstance(n, Neg):  return f"(- {afficher(n.expr)})"
    if isinstance(n, Bin):  return f"({n.op} {afficher(n.gauche)} {afficher(n.droi
   te)})"
```

Le même squelette sert à : évaluer, vérifier les types, compter, optimiser, compiler.

Le concept

Parcourir un arbre : le visiteur (suite)

Le **patron visiteur** formalise cela : une fonction par type de nœud, et un mécanisme de dispatch. Python moderne propose aussi la **correspondance de motifs** (`match`/`case`).

Les transformations

Une **transformation** construit un **nouvel arbre** à partir de l'ancien. Exemple essentiel : le **pliage de constantes** (*constant folding*) :

Avant	Après
`2 * 3 + x`	`6 + x`
`x * 1`	`x`
`x + 0`	`x`
`0 * f(x)`	**à ne pas simplifier** (si `f` a des effets de bord)

Le concept

Les transformations (suite)

Une optimisation doit **préserver le sens** du programme. Attention à `0 * f(x)` ou à la division par zéro `1 / 0` que l'on ne doit pas évaluer à la compilation.

Représentations de l'arbre

- **S-expression** : `(+ 2 (* 3 4))` : compacte, facile à comparer dans les tests.
- **Mermaid/DOT** : pour dessiner.
- **JSON** : pour échanger avec d'autres outils.

Pourquoi un AST plutôt que d'interpréter directement le texte ?

- on sépare la **structure** de son **traitement** (plusieurs traitements sur le même arbre) ;
- on peut **optimiser** avant d'exécuter ;
- on peut produire des **messages précis** avec les positions ;

Le concept

Pourquoi un AST plutôt que d'interpréter directement le texte ? (suite)

- les outils de qualité (analyseurs, formateurs, refactoring automatique) travaillent sur l'AST.

🧠 ***Astuce*** *: Python expose le sien : ast.parse("1 + 2"). Explore-le avec ast.dump pour voir un AST industriel.*

Exemple pas à pas

Problème : utiliser l'AST de HashScript pour (1) afficher en S-expression, (2) lister les variables, (3) plier les constantes.

On utilise le lexer et le parseur de référence (fournis dans `01-Course/code/hashscript.py`, construits dans les missions 2 et 3 puis étendus).

```
from hashscript import parser, Num, Var, Bin, Neg, Let, Print

def sexpr(n):
    if isinstance(n, Num):
        return str(n.valeur)
    if isinstance(n, Var):
        return n.nom
    if isinstance(n, Neg):
        return f"(- {sexpr(n.expr)})"
    if isinstance(n, Bin):
        return f"({n.op} {sexpr(n.gauche)} {sexpr(n.droite)})"
```

Exemple pas à pas (suite)

```
    raise TypeError(type(n).__name__)

def variables(n):
    if isinstance(n, Var):
        return {n.nom}
    if isinstance(n, Neg):
        return variables(n.expr)
    if isinstance(n, Bin):
        return variables(n.gauche) | variables(n.droite)
    return set()

programme = parser("let r = 2 * 3 + x * (4 - 4 + y);")
expression = programme[0].expr
print(sexpr(expression))          # (+ (* 2 3) (* x (+ (- 4 4) y)))
print(sorted(variables(expression)))     # ['x', 'y']
```

Le pliage de constantes : on reconstruit l'arbre en évaluant tout ce qui est constant.

Exemple pas à pas (suite)

```
import operator
from hashscript import parser, Num, Var, Bin, Neg

OPS = {"+": operator.add, "-": operator.sub, "*": operator.mul, "/": operator.true
   div}

def plier(n):
    if isinstance(n, Neg):
        e = plier(n.expr)
        return Num(-e.valeur) if isinstance(e, Num) else Neg(e)
    if not isinstance(n, Bin):
        return n
    g, d = plier(n.gauche), plier(n.droite)
    if n.op in OPS and isinstance(g, Num) and isinstance(d, Num):
        if n.op == "/" and d.valeur == 0:
            return Bin(n.op, g, d, n.ligne, n.col)          # on ne simplifie pas
    : l'erreur doit rester à l'exécution
        return Num(OPS[n.op](g.valeur, d.valeur))
    if n.op == "*" and isinstance(g, Num) and g.valeur == 1: return d
```

Exemple pas à pas (suite)

```
    if n.op == "*" and isinstance(d, Num) and d.valeur == 1: return g
    if n.op == "+" and isinstance(g, Num) and g.valeur == 0: return d
    if n.op == "+" and isinstance(d, Num) and d.valeur == 0: return g
    return Bin(n.op, g, d, n.ligne, n.col)

def sexpr(n):
    if isinstance(n, Num): return str(n.valeur)
    if isinstance(n, Var): return n.nom
    if isinstance(n, Neg): return f"(- {sexpr(n.expr)})"
    return f"({n.op} {sexpr(n.gauche)} {sexpr(n.droite)})"

for source in ("2 * 3 + x;", "x * 1 + 0;", "(4 - 4 + y) * 2;", "1 / 0 + x;"):
    arbre = parser(source)[0]
    print(f"{source:20s} → {sexpr(plier(arbre))}")
```

Lecture : `2 * 3 + x` devient `(+ 6 x)` ; `x * 1 + 0` devient `x` ; `1 / 0` n'est **pas** évalué à la compilation (on laisse l'erreur au moment de l'exécution, avec sa position). Chaque règle de simplification préserve le sens du programme.

À toi, avec aide

Écris `profondeur(n)` qui renvoie la **hauteur** d'un AST d'expression (un `Num` ou `Var` seul a la hauteur 1) et `compter(n)` qui renvoie le nombre de nœuds. Calcule les deux pour `1 + 2 * (3 - x)`.
Dessine ensuite l'arbre en Mermaid avec une fonction `vers_mermaid(n)` qui produit le texte d'un `flowchart TD`.

À toi, avec aide · réponse

Indices progressifs

Indice 1

`profondeur` d'un `Bin` = 1 + max(gauche, droite).

Indice 2

Numérote les nœuds avec un compteur pour créer des identifiants uniques.

Indice 3

Chaque `Bin` produit une arête vers ses deux fils.

À toi, avec aide · réponse

Réponse

```
from hashscript import parser, Num, Var, Bin, Neg

def profondeur(n):
    if isinstance(n, Bin):
        return 1 + max(profondeur(n.gauche), profondeur(n.droite))
    if isinstance(n, Neg):
        return 1 + profondeur(n.expr)
    return 1

def compter(n):
    if isinstance(n, Bin):
        return 1 + compter(n.gauche) + compter(n.droite)
    if isinstance(n, Neg):
        return 1 + compter(n.expr)
    return 1

def vers_mermaid(racine):
    lignes, compteur = ["flowchart TD"], [0]
    def visiter(n):
```

Réponse (suite)

```
        compteur[0] += 1
        ident = f"n{compteur[0]}"
        if isinstance(n, Bin):
            lignes.append(f'    {ident}["{n.op}"]')
            for fils in (n.gauche, n.droite):
                f_id = visiter(fils)
                lignes.append(f"    {ident} --> {f_id}")
        elif isinstance(n, Neg):
            lignes.append(f'    {ident}["-"]')
            lignes.append(f"    {ident} --> {visiter(n.expr)}")
        elif isinstance(n, Num):
            lignes.append(f'    {ident}["{n.valeur}"]')
        else:
            lignes.append(f'    {ident}["{n.nom}"]')
        return ident
    visiter(racine)
    return "\n".join(lignes)

arbre = parser("1 + 2 * (3 - x);")[0]
```

À toi, avec aide · réponse

Réponse (suite)

```
print(profondeur(arbre), compter(arbre))        # 4 7
print(vers_mermaid(arbre))
```

L'arbre a 7 nœuds (4 feuilles + 3 opérateurs) et une hauteur de 4.

À toi, en solo

Simplifications algébriques. Étends le pliage de constantes avec :

- `x - x` → `0` (même variable des deux côtés) ;
- `x * 0` → `0` **seulement si** `x` ne contient que des constantes et des variables (pas d'appel de fonction) ;
- `-(-x)` → `x` ;
- `x / 1` → `x`.

Écris les tests : chaque simplification doit conserver le résultat de l'évaluation. Vérifie-le avec un petit évaluateur, sur 300 expressions aléatoires dans lesquelles les variables prennent des valeurs aléatoires.

À toi, en solo · réponse

Pistes de correction

```
import operator, random
from hashscript import Num, Var, Bin, Neg

OPS = {"+": operator.add, "-": operator.sub, "*": operator.mul, "/": operator.true
   div}

def meme(a, b):
    return type(a) is type(b) and ((isinstance(a, Var) and a.nom == b.nom) or (isi
   nstance(a, Num) and a.valeur == b.valeur))

def pur(n):                                          # sans appel : ici toutes no
   s expressions le sont
    return True

def plier(n):
    if isinstance(n, Neg):
        e = plier(n.expr)
        if isinstance(e, Num): return Num(-e.valeur)
        if isinstance(e, Neg): return e.expr
```

À toi, en solo · réponse

Pistes de correction (suite)

```
        return Neg(e)
    if not isinstance(n, Bin):
        return n
    g, d = plier(n.gauche), plier(n.droite)
    if n.op in OPS and isinstance(g, Num) and isinstance(d, Num):
        if n.op == "/" and d.valeur == 0:
            return Bin(n.op, g, d)
        return Num(OPS[n.op](g.valeur, d.valeur))
    if n.op == "-" and meme(g, d): return Num(0)
    if n.op == "*" and ((isinstance(g, Num) and g.valeur == 0 and pur(d)) or (isin
   stance(d, Num) and d.valeur == 0 and pur(g))): return Num(0)
    if n.op == "*" and isinstance(g, Num) and g.valeur == 1: return d
    if n.op == "*" and isinstance(d, Num) and d.valeur == 1: return g
    if n.op == "/" and isinstance(d, Num) and d.valeur == 1: return g
    if n.op == "+" and isinstance(g, Num) and g.valeur == 0: return d
    if n.op == "+" and isinstance(d, Num) and d.valeur == 0: return g
    return Bin(n.op, g, d)

def evaluer(n, env):
```

Pistes de correction (suite)

```
    if isinstance(n, Num): return n.valeur
    if isinstance(n, Var): return env[n.nom]
    if isinstance(n, Neg): return -evaluer(n.expr, env)
    return OPS[n.op](evaluer(n.gauche, env), evaluer(n.droite, env))

def aleatoire(rng, profondeur=3):
    if profondeur == 0 or rng.random() < 0.25:
        return Num(rng.choice([0, 1, 2, 3])) if rng.random() < 0.5 else Var(rng.ch
   oice("xy"))
    if rng.random() < 0.15:
        return Neg(aleatoire(rng, profondeur - 1))
    return Bin(rng.choice(["+", "-", "*", "/"]), aleatoire(rng, profondeur - 1), a
   leatoire(rng, profondeur - 1))

rng = random.Random(11)
verifies = 0
for _ in range(300):
    arbre = aleatoire(rng)
    env = {"x": rng.randint(1, 9), "y": rng.randint(1, 9)}
```

Pistes de correction (suite)

```
    try:
        attendu = evaluer(arbre, env)
    except ZeroDivisionError:
        continue
    assert abs(evaluer(plier(arbre), env) - attendu) < 1e-9, arbre
    verifies += 1
print(verifies, "expressions vérifiées")
```

Le test aléatoire vérifie la propriété centrale d'une optimisation : **elle ne change pas le résultat**.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Optimiser en changeant le sens	`0 * f(x)` supprime un effet de bord	Ne simplifier que ce qui est pur et sûr
Évaluer `1 / 0` à la compilation	L'erreur change de moment ou disparaît	Laisser l'erreur à l'exécution
Garder le bruit syntaxique dans l'AST	Arbres lourds, traitements compliqués	Abstraire les parenthèses, virgules, séparateurs
Modifier l'arbre sur place	Effets de bord imprévus	Construire un nouvel arbre

Mini-boss

Boîte à outils d'AST. À partir des programmes HashScript analysés avec `parser`, construis un module `outils_ast.py` avec :

1. un **visiteur générique** `parcourir(noeud, fonction)` qui appelle `fonction` sur chaque nœud (instructions et expressions, y compris `Si`, `TantQue`, `Fun`) ;
2. `variables_lues(programme)`, `variables_ecrites(programme)` (déclarées ou affectées), `fonctions_appelees(programme)` ;
3. `statistiques(programme)` : nombre de nœuds par type, profondeur maximale ;
4. une **transformation** `renommer(programme, ancien, nouveau)` qui renomme une variable en respectant les portées simples (ne touche pas une variable de même nom déclarée dans une fonction qui la masque) ;
5. un **formateur** qui reconstitue du texte HashScript lisible depuis l'AST (`vers_texte`), et la propriété : `parser(vers_texte(parser(s)))` produit le même AST que `parser(s)` sur 20 programmes.

Mini-boss · réponse

Pistes de correction

Un bon travail : `parcourir` utilise `dataclasses.fields` pour descendre dans les enfants (listes ou nœuds) sans écrire un `if` par type; `statistiques` compte avec un `Counter` et `max` sur les profondeurs; `renommer` maintient une pile de noms masqués (dès qu'une `Fun` déclare un paramètre du même nom, on ne renomme pas dans son corps); `vers_texte` gère l'indentation des blocs et ajoute des parenthèses seulement quand la priorité l'exige. Le test de ronde (« aller-retour ») compare les AST avec l'égalité des dataclasses, en ignorant les champs de position (ligne/col).

Quiz éclair

Quiz éclair

1. L'AST garde :

A. la structure essentielle, sans le bruit syntaxique
B. tous les espaces et commentaires
C. seulement les nombres

2. Le pliage de constantes consiste à :

A. supprimer les variables
B. évaluer à l'avance les calculs constants
C. traduire en assembleur

3. Une optimisation doit toujours :

A. réduire le nombre de lignes

Quiz éclair

Quiz éclair (suite)

B. utiliser plus de mémoire
C. préserver le sens du programme

4. 🔁 *Révision mission 03* : Dans la descente récursive, on écrit :

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

5. 🔁 *Révision mission 03* : La boucle `( … )*` dans une règle produit :

A. une associativité à droite
B. une associativité à gauche
C. aucune associativité

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Quel autre document ou outil de ta vie existe à la fois en « texte » et en « structure » (une recette, une formule, un plan) ?
2. Pourquoi vaut-il mieux transformer un arbre en construisant un nouvel arbre qu'en le modifiant sur place ?

À retenir

À retenir

1. AST = structure essentielle du programme.
2. Nœuds typés (dataclasses) + parcours récursif.
3. Transformation = nouvel arbre, sens préservé.
4. S-expression, Mermaid, JSON pour représenter.
5. Un test aléatoire vérifie qu'une optimisation ne change pas le résultat.

MISSION ACCOMPLIE

+100 XP

Badge débloqué : Jardinier d'arbres

Prochaine mission : L'analyse sémantique

