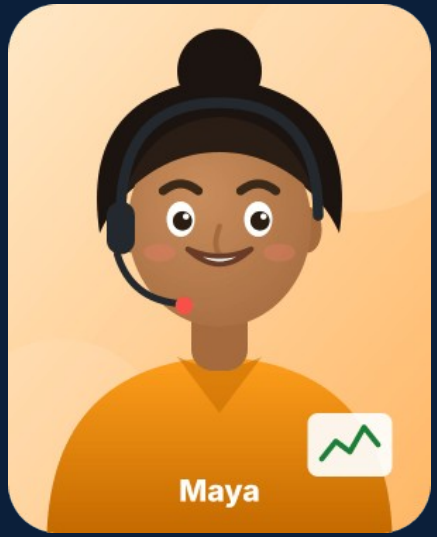
HEF-090 · MISSION 06 / 10

L'interpréteur : faire vivre l'arbre

Un arbre vérifié ne fait encore rien. L'interpréteur le parcourt et exécute chaque nœud.





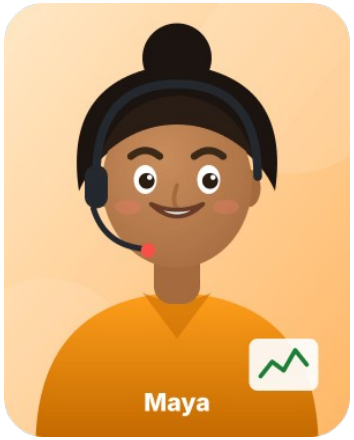
Badge : Chef d'exécution

90 min · intermédiaire

L'histoire

HashScript sait lire, comprendre et vérifier un programme. Reste à l'exécuter. Kodi a une intuition : « Il suffit de parcourir l'arbre et de faire ce qu'il dit. » Sam sourit : « C'est exactement ça. Chaque type de nœud sait comment s'exécuter. »

Tu écris l'interpréteur de HashScript : variables, calculs, conditions, boucles.



Ta mission

À la fin, je serai capable de…

1. parcourir un AST pour l'exécuter (interpréteur « tree-walking »)
2. gérer un environnement de variables avec des portées de blocs
3. exécuter instructions, expressions, conditions et boucles
4. signaler les erreurs d'exécution avec leur position
5. protéger l'interpréteur contre les boucles infinies

Défi d'échauffement

Défi d'échauffement

Tu reçois une recette sous forme d'arbre (étapes imbriquées). Pour la réaliser, tu fais le tour de l'arbre : sur chaque étape, tu **fais** l'action correspondante, et tu descends dans ses sous-étapes. Comment sais-tu quand arrêter ? Que fais-tu quand une étape est une répétition (« recommence 3 fois ») ou une condition (« si la sauce a épaissi ») ?

Le concept

Le principe d'un interpréteur d'arbre

Un **interpréteur** exécute un programme **directement** à partir de son AST. Il comporte deux fonctions principales :

- `evaluer(expression)` : **calcule** une valeur ;
- `executer(instruction)` : **fait** quelque chose (affecter, afficher, répéter).

Le travail consiste à regarder le **type de nœud** et à appliquer la règle correspondante. On appelle cela l'**évaluation récursive**.

Nœud	Règle d'exécution
`Num`, `Bool`	renvoyer la valeur
`Var`	lire la variable dans l'environnement

Le concept

Le principe d'un interpréteur d'arbre (suite)

Nœud	Règle d'exécution
`Bin(op, g, d)`	évaluer `g`, évaluer `d`, appliquer `op`
`Neg(e)`	évaluer `e`, changer le signe
`Let(nom, e)`	évaluer `e`, **créer** la variable
`Assign(nom, e)`	évaluer `e`, **modifier** la variable existante
`Print(e)`	évaluer `e`, ajouter à la sortie
`Si(c, alors, sinon)`	évaluer `c` ; exécuter le bon bloc
`TantQue(c, corps)`	tant que `c` est vrai, exécuter le corps

L'environnement et les portées

Le concept

L'environnement et les portées (suite)

Les variables vivent dans un **environnement** : une pile de dictionnaires, un par bloc. Entrer dans un bloc (`{ … }`) empile un dictionnaire ; en sortir le dépile.

- `let` **crée** une variable dans la portée **courante** ;
- lire ou modifier une variable cherche **du plus interne au plus externe**.

Valeurs et types à l'exécution

HashScript manipule des nombres (entiers/flottants Python) et des booléens. Les erreurs d'exécution typiques : division par zéro, variable inconnue, (plus tard) mauvais appel de fonction. On les signale avec la **position du nœud** fautif.

Les gardes-fous

Le concept

Les gardes-fous (suite)

Un interpréteur peut être piégé par `while true { }`. On borne donc le nombre de pas : au-delà d'une limite, on lève une erreur claire. De même, une récursion trop profonde doit être interceptée.

L'interpréteur en Python

```
import operator
from hashscript import parser, Let, Assign, Print, Si, TantQue, Num, Bool, Var, Bi
   n, Neg, ErreurLangage

OPS = {"+": operator.add, "-": operator.sub, "*": operator.mul, "/": operator.true
   div,
       "<": operator.lt, ">": operator.gt, "<=": operator.le, ">=": operator.ge,
       "==": operator.eq, "!=": operator.ne}

class Interprete:
    def __init__(self):
        self.portees, self.sortie = [{}], []
```

Le concept

L'interpréteur en Python (suite)

```
    def chercher(self, nom, noeud):
        for portee in reversed(self.portees):
            if nom in portee:
                return portee
        raise ErreurLangage(f"variable inconnue {nom!r}", noeud.ligne, noeud.col)

    def evaluer(self, n):
        if isinstance(n, (Num, Bool)):
            return n.valeur
        if isinstance(n, Var):
            return self.chercher(n.nom, n)[n.nom]
        if isinstance(n, Neg):
            return -self.evaluer(n.expr)
        if isinstance(n, Bin):
            a, b = self.evaluer(n.gauche), self.evaluer(n.droite)
            if n.op == "/" and b == 0:
                raise ErreurLangage("division par zéro", n.ligne, n.col)
            return OPS[n.op](a, b)
```

Le concept

L'interpréteur en Python (suite)

```
        raise ErreurLangage(f"expression non supportée : {type(n).__name__}", 0, 0
    )

    def bloc(self, instructions):
        self.portees.append({})
        try:
            for i in instructions:
                self.executer(i)
        finally:
            self.portees.pop()

    def executer(self, n):
        if isinstance(n, Let):
            self.portees[-1][n.nom] = self.evaluer(n.expr)
        elif isinstance(n, Assign):
            valeur = self.evaluer(n.expr)
            self.chercher(n.nom, n)[n.nom] = valeur
        elif isinstance(n, Print):
            self.sortie.append(self.formater(self.evaluer(n.expr)))
```

Le concept

L'interpréteur en Python (suite)

```
        elif isinstance(n, Si):
            self.bloc(n.alors if self.evaluer(n.cond) else n.sinon)
        elif isinstance(n, TantQue):
            while self.evaluer(n.cond):
                self.bloc(n.corps)
        else:
            self.evaluer(n)

    @staticmethod
    def formater(v):
        if v is True: return "true"
        if v is False: return "false"
        if isinstance(v, float) and v == int(v): return str(int(v))
        return str(v)

    def lancer(self, source):
        for instruction in parser(source):
            self.executer(instruction)
        return self.sortie
```

Le concept

L'interpréteur en Python (suite)

🧠 ***Astuce*** *: isinstance en cascade est la forme la plus simple. Dans un langage plus grand, on utilise le patron visiteur ou une table type de nœud → méthode.*

Exemple pas à pas

Problème : exécuter des programmes HashScript avec l'interpréteur et vérifier le comportement.

```
import operator
from hashscript import parser, Let, Assign, Print, Si, TantQue, Num, Bool, Var, Bi
   n, Neg, ErreurLangage

OPS = {"+": operator.add, "-": operator.sub, "*": operator.mul, "/": operator.true
   div,
       "<": operator.lt, ">": operator.gt, "<=": operator.le, ">=": operator.ge,
       "==": operator.eq, "!=": operator.ne}

class Interprete:
    def __init__(self):
        self.portees, self.sortie = [{}], []

    def chercher(self, nom, noeud):
        for portee in reversed(self.portees):
            if nom in portee:
```

Exemple pas à pas

Exemple pas à pas (suite)

```
                return portee
        raise ErreurLangage(f"variable inconnue {nom!r}", noeud.ligne, noeud.col)

    def evaluer(self, n):
        if isinstance(n, (Num, Bool)): return n.valeur
        if isinstance(n, Var): return self.chercher(n.nom, n)[n.nom]
        if isinstance(n, Neg): return -self.evaluer(n.expr)
        if isinstance(n, Bin):
            a, b = self.evaluer(n.gauche), self.evaluer(n.droite)
            if n.op == "/" and b == 0:
                raise ErreurLangage("division par zéro", n.ligne, n.col)
            return OPS[n.op](a, b)
        raise ErreurLangage(f"expression non supportée : {type(n).__name__}", 0, 0
    )

    def bloc(self, instructions):
        self.portees.append({})
        try:
            for i in instructions:
```

Exemple pas à pas

Exemple pas à pas (suite)

```
                self.executer(i)
        finally:
            self.portees.pop()

    def executer(self, n):
        if isinstance(n, Let): self.portees[-1][n.nom] = self.evaluer(n.expr)
        elif isinstance(n, Assign):
            valeur = self.evaluer(n.expr)
            self.chercher(n.nom, n)[n.nom] = valeur
        elif isinstance(n, Print): self.sortie.append(self.formater(self.evaluer(n
    .expr)))
        elif isinstance(n, Si): self.bloc(n.alors if self.evaluer(n.cond) else n.s
    inon)
        elif isinstance(n, TantQue):
            while self.evaluer(n.cond):
                self.bloc(n.corps)
        else: self.evaluer(n)

    @staticmethod
```

Exemple pas à pas (suite)

```
    def formater(v):
        if v is True: return "true"
        if v is False: return "false"
        if isinstance(v, float) and v == int(v): return str(int(v))
        return str(v)

    def lancer(self, source):
        for instruction in parser(source):
            self.executer(instruction)
        return self.sortie

programmes = {
    "somme 1..100": ("let i = 1; let s = 0; while i <= 100 { s = s + i; i = i + 1;
    } print s;", ["5050"]),
    "puissance 2^10": ("let r = 1; let k = 0; while k < 10 { r = r * 2; k = k + 1;
    } print r;", ["1024"]),
    "Fibonacci(10)": ("let a = 0; let b = 1; let n = 0; while n < 10 { let t = a +
    b; a = b; b = t; n = n + 1; } print a;", ["55"]),
    "conditions": ("let x = 7; if x > 5 { print 1; } else { print 2; } if x < 5 {
```

Exemple pas à pas (suite)

```
    print 3; } else { print 4; }", ["1", "4"]),
     "portée de bloc": ("let x = 1; if true { let x = 2; print x; } print x;", ["2"
    , "1"]),
}
for nom, (source, attendu) in programmes.items():
    sortie = Interprete().lancer(source)
    assert sortie == attendu, (nom, sortie)
    print(f"{nom:16s} → {sortie}")

for source in ("print x;", "let a = 1 / (2 - 2);"):
    try:
        Interprete().lancer(source)
    except ErreurLangage as e:
        print(f"erreur ligne {e.ligne}, colonne {e.colonne} : {e.message}")
```

Lecture :

- chaque programme s'exécute en parcourant l'AST ; le résultat est comparé à la valeur attendue ;

Exemple pas à pas (suite)

- `let x = 2` dans le bloc `if` **masque** le `x` extérieur (portée de bloc) mais ne le modifie pas : le second `print x` affiche 1 ;
- les erreurs d'exécution portent la **position** du nœud fautif (colonne de l'opérateur `/`, de la variable inconnue).

À toi, avec aide

Ajoute à l'interpréteur une **limite de pas** : `Interprete(limite=10000)` compte les instructions exécutées et lève `ErreurLangage("trop d'instructions…")` au-delà. Montre qu'elle arrête `while true { }`, et qu'un programme normal (somme de 1 à 100) passe avec une limite de 10 000. Pense à ce que l'on compte dans une boucle dont le corps est vide.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Incrémente un compteur dans `executer` **et** à chaque tour de `TantQue`.

Indice 2

Une boucle `while true { }` n'exécute aucune instruction dans son corps : seul le comptage par tour l'arrête.

Indice 3

Réinitialise le compteur dans `lancer`.

À toi, avec aide · réponse

Réponse

```
# (classe Interprete de l'exemple, étendue)
class InterpreteLimite(Interprete):
    def __init__(self, limite=10000):
        super().__init__()
        self.limite, self.pas = limite, 0

    def compter(self):
        self.pas += 1
        if self.pas > self.limite:
            raise ErreurLangage("trop d'instructions exécutées (boucle infinie ?)"
   , 0, 0)

    def executer(self, n):
        self.compter()
        if isinstance(n, TantQue):
            while self.evaluer(n.cond):
                self.compter()                  # compte aussi les tours d'une bo
   ucle au corps vide
                self.bloc(n.corps)
```

Réponse (suite)

```
        else:
            super().executer(n)

assert InterpreteLimite(10000).lancer("let i = 1; let s = 0; while i <= 100 { s =
   s + i; i = i + 1; } print s;") == ["5050"]
try:
    InterpreteLimite(1000).lancer("while true { }")
    raise AssertionError("boucle infinie non arrêtée")
except ErreurLangage as e:
    print(e.message)
```

La classe dérivée réutilise tout l'interpréteur et ne surcharge que `executer` pour compter et traiter `TantQue`.

À toi, en solo

Trace et profil d'exécution. Étends l'interpréteur pour produire :

1. une **trace** : la liste des `(ligne, type de nœud)` des instructions exécutées (affiche les 10 premières pour `somme 1..5`) ;
2. un **profil** : un `Counter` du nombre d'exécutions par type de nœud, y compris les expressions (`Bin`, `Var`, `Num`) ;
3. une option `pas_a_pas` qui appelle une fonction de rappel avant chaque instruction (par exemple pour afficher l'état des variables).

Montre sur le programme « somme 1..10 » que `Bin` est le nœud le plus exécuté. Qu'en conclus-tu sur les optimisations possibles ?

À toi, en solo · réponse

Pistes de correction

```
# (classe Interprete de l'exemple)
from collections import Counter

class InterpreteTrace(Interprete):
    def __init__(self, rappel=None):
        super().__init__()
        self.trace, self.profil, self.rappel = [], Counter(), rappel

    def evaluer(self, n):
        self.profil[type(n).__name__] += 1
        return super().evaluer(n)

    def executer(self, n):
        self.profil[type(n).__name__] += 1
        self.trace.append((getattr(n, "ligne", 0), type(n).__name__))
        if self.rappel:
            self.rappel(n, self.portees)
        super().executer(n)
```

À toi, en solo · réponse

Pistes de correction (suite)

```
interp = InterpreteTrace()
interp.lancer("let i = 1; let s = 0; while i <= 10 { s = s + i; i = i + 1; } print
    s;")
print(interp.trace[:10])
print(interp.profil.most_common())
etats = []
InterpreteTrace(rappel=lambda n, portees: etats.append(dict(portees[-1]))).lancer(
   "let a = 1; a = a + 1; print a;")
print(etats)
```

Dans le profil, `Bin`, `Var` et `Num` dominent largement : l'essentiel du travail est du calcul d'expressions. Le **pliage de constantes**, la **réduction de l'allocation d'environnements** et la **compilation vers du bytecode** (mission 8) s'attaquent à cela.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Oublier de dépiler la portée en cas d'erreur	Les variables fuient dans le bloc suivant	`try/finally` autour du bloc
Évaluer deux fois une expression	Effets de bord dupliqués	Évaluer une fois, stocker le résultat
Pas de limite de pas	Un programme malveillant bloque la machine	Compteur de pas et de profondeur
Erreurs sans position	Impossible à localiser	Chaque nœud garde ligne et colonne

Mini-boss

Mini-boss

Fonctions natives (built-ins). Ajoute à l'interpréteur des fonctions fournies par le langage, appelables comme `abs(-3)` ou `max(2, 7)` : `abs`, `max`, `min`, `racine` (racine carrée), `entier` (partie entière), `puissance(a, b)`.

1. Le parseur sait déjà analyser les appels (`Appel(fonction=Var, args=[…])`) ; l'interpréteur doit les évaluer.
2. Chaque fonction native a un nombre d'arguments attendu ; une erreur claire (avec position) est levée sinon.
3. `racine` d'un nombre négatif doit lever une erreur d'exécution positionnée.
4. Une fonction native peut être utilisée dans une expression (`1 + max(2, 3) * 2`).
5. Teste avec au moins 12 programmes, valides et invalides.

Mini-boss · réponse

Pistes de correction

```
import math
# (classe Interprete de l'exemple)
from hashscript import Appel

class InterpreteNatives(Interprete):
    NATIVES = {
        "abs": (1, abs), "max": (2, max), "min": (2, min),
        "racine": (1, math.sqrt), "entier": (1, math.floor), "puissance": (2, pow)
   ,
    }

    def evaluer(self, n):
        if isinstance(n, Appel) and isinstance(n.fonction, Var) and n.fonction.nom
    in self.NATIVES:
            arite, f = self.NATIVES[n.fonction.nom]
            if len(n.args) != arite:
                raise ErreurLangage(f"{n.fonction.nom} attend {arite} argument(s),
    reçu {len(n.args)}", n.ligne, n.col)
            valeurs = [self.evaluer(a) for a in n.args]
```

Mini-boss · réponse

Pistes de correction (suite)

```
            if n.fonction.nom == "racine" and valeurs[0] < 0:
                raise ErreurLangage("racine d'un nombre négatif", n.ligne, n.col)
            return f(*valeurs)
        if isinstance(n, Appel):
            raise ErreurLangage(f"fonction inconnue {getattr(n.fonction, 'nom', '?
   ')!r}", n.ligne, n.col)
        return super().evaluer(n)

def lancer(source):
    return InterpreteNatives().lancer(source)

assert lancer("print abs(-3); print max(2, 7); print min(2, 7);") == ["3", "7", "2
   "]
assert lancer("print 1 + max(2, 3) * 2;") == ["7"]
assert lancer("print racine(16); print entier(3.7); print puissance(2, 10);") == [
   "4", "3", "1024"]
for mauvais, msg in (("print max(1);", "attend 2"), ("print racine(0 - 4);", "néga
   tif"), ("print inconnue(1);", "fonction inconnue")):
    try:
```

Mini-boss · réponse

Pistes de correction (suite)

```
        lancer(mauvais)
        raise AssertionError(mauvais)
    except ErreurLangage as e:
        assert msg in e.message, e.message
print("fonctions natives ok")
```

Une table `nom → (arité, fonction Python)` rend l'ajout d'une fonction native trivial : c'est la base d'une « bibliothèque standard » minimale.

Quiz éclair

Quiz éclair

1. Un interpréteur d'arbre exécute un programme en :

A. le traduisant en code machine à l'avance
B. parcourant l'AST et appliquant une règle par type de nœud
C. relisant le texte caractère par caractère

2. Entrer dans un bloc `{ … }` correspond à :

A. supprimer les variables
B. terminer le programme
C. empiler une nouvelle portée

3. Une limite de pas protège contre :

A. les boucles infinies

Quiz éclair (suite)

B. les erreurs de syntaxe
C. les variables inutilisées

4. 🔁 *Révision mission 05* : L'analyse sémantique vérifie :

A. uniquement les parenthèses
B. noms, déclarations et types
C. la vitesse d'exécution

5. 🔁 *Révision mission 05* : Le typage statique vérifie les types :

A. pendant l'exécution seulement
B. jamais
C. avant l'exécution

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Dans ta vie, où « exécutes-tu » une structure en la parcourant (une checklist, un organigramme, un arbre de décision) ?
2. Pourquoi mettre des gardes-fous dans un interpréteur avant de l'ouvrir à d'autres utilisateurs ?

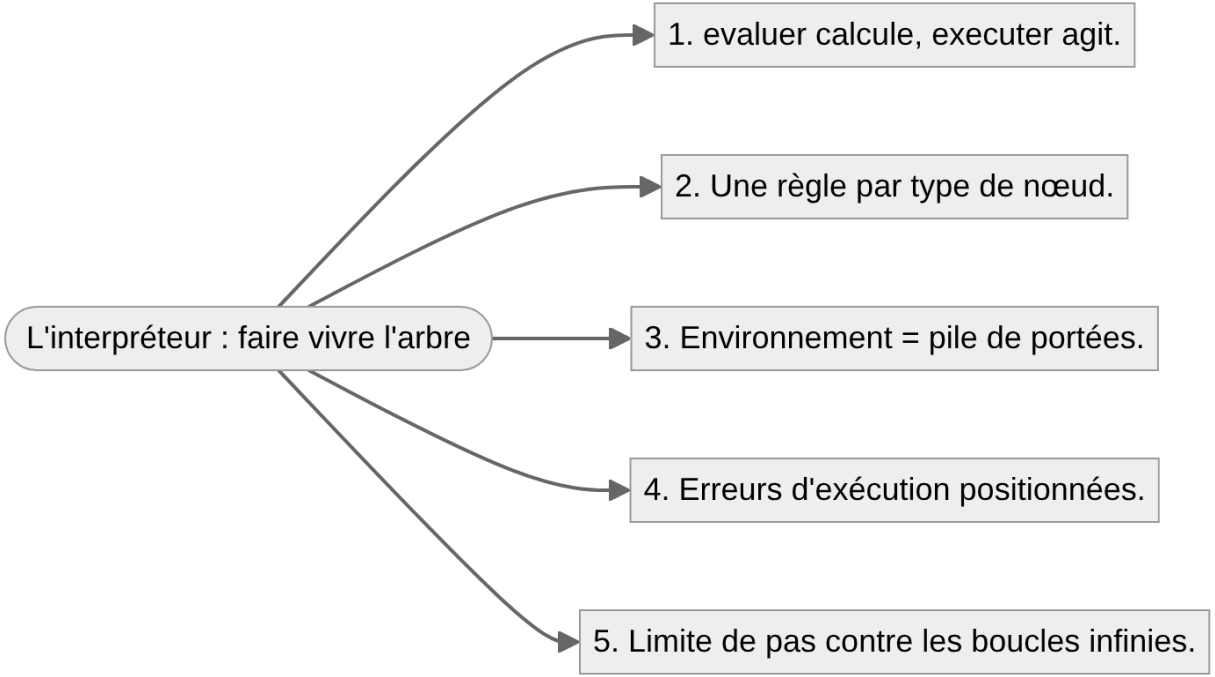
À retenir

À retenir

1. evaluer calcule, executer agit.
2. Une règle par type de nœud.
3. Environnement = pile de portées.
4. Erreurs d'exécution positionnées.
5. Limite de pas contre les boucles infinies.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Chef d'exécution

Prochaine mission : Environnement, portée et fermetures

