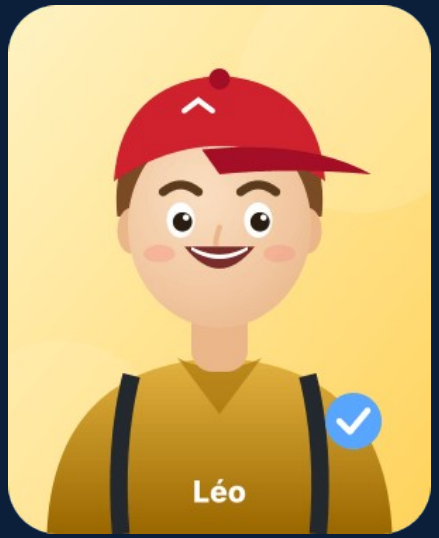
HEF-090 · MISSION 08 / 10

Bytecode et machine virtuelle

Au lieu de parcourir l'arbre à chaque fois, on le traduit une fois en instructions simples exécutées par une petite machine.



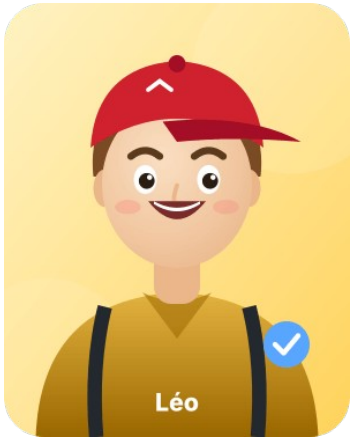


Badge : Pilote de VM

100 min · intermédiaire

L'histoire

La boucle de HashScript qui additionne 1 à 100 000 prend plusieurs secondes. Sam profile : le parcours de l'arbre, avec ses `isinstance` en cascade, est le goulot.



> *« Python, Java et Lua ne parcourent pas l'arbre, dit-il. Ils le traduisent d'abord en bytecode, une suite d'instructions très simples, qu'une machine virtuelle exécute en boucle. C'est plus rapide, plus compact, et ça ouvre la porte aux optimisations. »*

Tu compiles HashScript vers un bytecode à pile et tu écris sa machine virtuelle.

Ta mission

À la fin, je serai capable de…

1. expliquer ce qu'est un bytecode et une machine virtuelle à pile
2. compiler expressions et instructions vers du bytecode
3. gérer les sauts conditionnels et les boucles (étiquettes, remplissage a posteriori)
4. écrire une VM et un désassembleur
5. mesurer le gain par rapport à l'interpréteur d'arbre

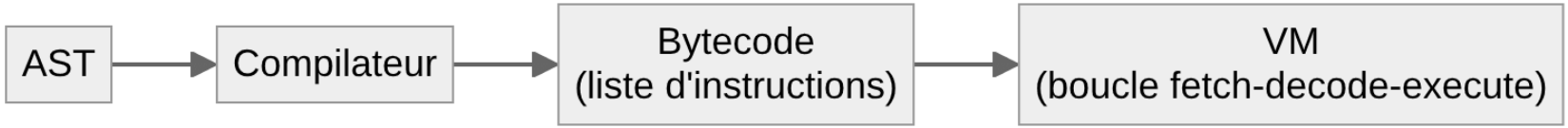
Défi d'échauffement

Tu veux calculer `(2 + 3) * 4` avec une calculatrice à **pile** (notation polonaise inverse) : tu tapes `2`, `3`, `+`, `4`, `×`. Écris la suite de touches. Ensuite : comment traduirais-tu `si a < b alors ... sinon ...` avec seulement des instructions simples du type « va à la ligne N » et « va à la ligne N si le dernier résultat est faux » ?

Le concept

Du parcours d'arbre au bytecode

Un **interpréteur d'arbre** repasse sur la même structure à chaque exécution. Un **compilateur vers bytecode** la parcourt **une seule fois** et produit une **liste plate d'instructions** simples, qu'une **machine virtuelle** (VM) exécute dans une boucle serrée.



Description du schéma : l'AST est traduit par le compilateur en bytecode, qui est exécuté par la VM. C'est la même boucle fetch-decode-execute que celle d'un processeur (HEF-040), en logiciel.

Le concept

La machine à pile

La VM dispose d'une **pile** d'opérandes et d'un **compteur de programme** (PC). Les instructions travaillent sur la pile :

Instruction	Effet
`PUSH v`	empile la valeur v
`LOAD x`	empile la valeur de la variable x
`STORE x`	dépile et range dans la variable x
`ADD`, `SUB`, `MUL`, `DIV`	dépile deux valeurs, empile le résultat
`LT`, `GT`, `LE`, `GE`, `EQ`, `NE`	comparaisons (empile un booléen)
`NEG`	change le signe

Le concept

La machine à pile (suite)

Instruction	Effet
`PRINT`	dépile et affiche
`JMP c`	saute à l'instruction c
`JZ c`	dépile ; si faux, saute à c
`POP`	dépile et jette
`HALT`	arrête

`2 + 3 * 4` se compile en : `PUSH 2, PUSH 3, PUSH 4, MUL, ADD`. L'**ordre post-fixé** vient directement du parcours de l'arbre « fils gauche, fils droit, puis le nœud ».

Compiler les instructions de contrôle

Le concept

Compiler les instructions de contrôle (suite)

```
si c { A } sinon { B }

    <c>
    JZ  vers_sinon
    <A>
    JMP vers_fin
vers_sinon: <B>
vers_fin: ...
```

```
tantque c { C }

début : <c>
      JZ  vers_fin
      <C>
      JMP début
vers_fin: ...
```

Problème : on ne connaît pas l'adresse de `vers_fin` quand on émet le saut. Solution : **remplissage a posteriori** (*backpatching*) : on émet `JZ ?`, on garde son indice, et on le complète une fois la cible connue.

La VM

Le concept

La VM (suite)

```
def executer_vm(code):
    pile, variables, sortie, pc = [], {}, [], 0
    while True:
        op, *arg = code[pc]
        pc += 1
        if op == "PUSH": pile.append(arg[0])
        elif op == "LOAD": pile.append(variables[arg[0]])
        elif op == "STORE": variables[arg[0]] = pile.pop()
        elif op in OPS_BIN:
            b = pile.pop(); a = pile.pop()
            pile.append(OPS_BIN[op](a, b))
        elif op == "JMP": pc = arg[0]
        elif op == "JZ":
            if not pile.pop(): pc = arg[0]
        elif op == "PRINT": sortie.append(formater(pile.pop()))
        elif op == "HALT": return sortie
```

Le concept

Pourquoi c'est plus rapide ?

- plus de `isinstance` en cascade : un simple dispatch sur un code d'opération ;
- structures plates, bonne localité de cache (HEF-040) ;
- le bytecode peut être **optimisé** (pliage, suppression des instructions inutiles) et **sérialisé** (fichiers `.pyc`) ;
- on peut aller plus loin : compilation **JIT** vers du code machine.

Voir le bytecode de Python

```
import dis

def somme(n):
    s = 0
    i = 1
```

Le concept

Voir le bytecode de Python (suite)

```
    while i <= n:
        s = s + i
        i = i + 1
    return s

dis.dis(somme)
```

On y retrouve `LOAD_FAST`, `STORE_FAST`, `BINARY_OP`, `COMPARE_OP`, `POP_JUMP_IF_FALSE`, `JUMP_BACKWARD`: la même famille d'instructions que notre mini VM.

Limites de notre version

Variables globales dans un seul dictionnaire (pas de portées de blocs), pas de fonctions. Les VM réelles ajoutent des **cadres d'appel** (une pile d'appels), des variables locales indexées, des objets et un ramasse-miettes.

Exemple pas à pas

Problème : compiler HashScript vers du bytecode, l'exécuter, le désassembler et mesurer le gain.

```
import operator, time
from hashscript import parser, executer, Let, Assign, Print, Si, TantQue, Num, Boo
   l, Var, Bin, Neg

NOMS_BIN = {"+": "ADD", "-": "SUB", "*": "MUL", "/": "DIV", "<": "LT", ">": "GT",
   "<=": "LE", ">=": "GE", "==": "EQ", "!=": "NE"}
OPS_BIN = {"ADD": operator.add, "SUB": operator.sub, "MUL": operator.mul, "DIV": o
   perator.truediv,
           "LT": operator.lt, "GT": operator.gt, "LE": operator.le, "GE": operator
   .ge, "EQ": operator.eq, "NE": operator.ne}

class Compilateur:
    def __init__(self):
        self.code = []

    def emettre(self, *instruction):
```

Exemple pas à pas (suite)

```
        self.code.append(tuple(instruction))
        return len(self.code) - 1

    def expr(self, n):
        if isinstance(n, (Num, Bool)): self.emettre("PUSH", n.valeur)
        elif isinstance(n, Var): self.emettre("LOAD", n.nom)
        elif isinstance(n, Neg): self.expr(n.expr); self.emettre("NEG")
        elif isinstance(n, Bin):
            self.expr(n.gauche); self.expr(n.droite); self.emettre(NOMS_BIN[n.op])
        else: raise NotImplementedError(type(n).__name__)

    def instr(self, n):
        if isinstance(n, (Let, Assign)):
            self.expr(n.expr); self.emettre("STORE", n.nom)
        elif isinstance(n, Print):
            self.expr(n.expr); self.emettre("PRINT")
        elif isinstance(n, Si):
            self.expr(n.cond)
            saut_faux = self.emettre("JZ", None)                  # cible inconnue
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    pour l'instant
            for i in n.alors: self.instr(i)
            if n.sinon:
                saut_fin = self.emettre("JMP", None)
                self.code[saut_faux] = ("JZ", len(self.code))      # remplissage a
    posteriori
                for i in n.sinon: self.instr(i)
                self.code[saut_fin] = ("JMP", len(self.code))
            else:
                self.code[saut_faux] = ("JZ", len(self.code))
        elif isinstance(n, TantQue):
            debut = len(self.code)
            self.expr(n.cond)
            sortie = self.emettre("JZ", None)
            for i in n.corps: self.instr(i)
            self.emettre("JMP", debut)
            self.code[sortie] = ("JZ", len(self.code))
        else:
            self.expr(n); self.emettre("POP")
```

Exemple pas à pas (suite)

```
    def compiler(self, programme):
        for i in programme: self.instr(i)
        self.emettre("HALT")
        return self.code

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 executer_vm(code):
    pile, variables, sortie, pc = [], {}, [], 0
    while True:
        op, *arg = code[pc]
        pc += 1
        if op == "PUSH": pile.append(arg[0])
        elif op == "LOAD": pile.append(variables[arg[0]])
```

Exemple pas à pas

Exemple pas à pas (suite)

```
        elif op == "STORE": variables[arg[0]] = pile.pop()
        elif op in OPS_BIN:
            b = pile.pop(); a = pile.pop()
            pile.append(OPS_BIN[op](a, b))
        elif op == "NEG": pile.append(-pile.pop())
        elif op == "JMP": pc = arg[0]
        elif op == "JZ":
            if not pile.pop(): pc = arg[0]
        elif op == "PRINT": sortie.append(formater(pile.pop()))
        elif op == "POP": pile.pop()
        elif op == "HALT": return sortie

def desassembler(code):
    return "\n".join(f"{i:3d}  {' '.join(str(x) for x in instr)}" for i, instr in
   enumerate(code))

def compiler_source(source):
    return Compilateur().compiler(parser(source))
```

Exemple pas à pas (suite)

```
source = "let i = 1; let s = 0; while i <= 5 { s = s + i; i = i + 1; } print s;"
code = compiler_source(source)
print(desassembler(code))
assert executer_vm(code) == executer(source) == ["15"]

# Comparaison de vitesse (même résultat, deux modes d'exécution)
long = "let i = 1; let s = 0; while i <= 20000 { s = s + i * 2; i = i + 1; } print
    s;"
code_long = compiler_source(long)
t = time.perf_counter(); r1 = executer(long); t_arbre = time.perf_counter() - t
t = time.perf_counter(); r2 = executer_vm(code_long); t_vm = time.perf_counter() -
    t
assert r1 == r2
print(f"arbre : {t_arbre:.3f} s ; VM : {t_vm:.3f} s ; gain ≈ {t_arbre / t_vm:.1f}x
   ")
```

Extrait du désassemblage :

Exemple pas à pas (suite)

```
 0  PUSH 1
 1  STORE i
 2  PUSH 0
 3  STORE s
 4  LOAD i          ← début de la boucle
 5  PUSH 5
 6  LE
 7  JZ 17           ← sortie de la boucle
 ...
16  JMP 4           ← retour au test
17  LOAD s
18  PRINT
19  HALT
```

Lecture : la boucle devient un `JMP` en arrière et un `JZ` en avant ; le résultat est **identique** à celui de l'interpréteur d'arbre, mais la VM évite le parcours récursif et les `isinstance` à chaque nœud.

À toi, avec aide

Compile à la main (puis vérifie avec le compilateur) :

```
let x = 3;
if x > 2 { print 1; } else { print 2; }
```

1. Écris le bytecode attendu avec les numéros d'instruction et les cibles de saut.
2. Exécute-le sur la VM et trace l'état de la pile à chaque instruction pour le début du programme.

À toi, avec aide · réponse

Indices progressifs

Indice 1

`let x = 3;` → `PUSH 3`, `STORE x`.

Indice 2

La condition `x > 2` → `LOAD x`, `PUSH 2`, `GT`, puis `JZ vers_sinon`.

Indice 3

Le bloc alors se termine par `JMP vers_fin` pour ne pas exécuter le sinon.

À toi, avec aide · réponse

Réponse

```
  0  PUSH 3
  1  STORE x
  2  LOAD x
  3  PUSH 2
  4  GT
  5  JZ 9          # si faux : aller au bloc sinon
  6  PUSH 1
  7  PRINT
  8  JMP 11        # fin du alors : sauter le sinon
  9  PUSH 2
 10  PRINT
 11  HALT
```

```
# (Compilateur et VM de l'exemple)
code = compiler_source("let x = 3; if x > 2 { print 1; } else { print 2; }")
print(desassembler(code))
assert executer_vm(code) == ["1"]
```

À toi, avec aide · réponse

Réponse (suite)

Pile pendant les premières instructions : `[]` → `[3]` (PUSH 3) → `[]` (STORE x) → `[3]` (LOAD x) → `[3, 2]` (PUSH 2) → `[True]` (GT) → `[]` (JZ : la condition est vraie, pas de saut).

À toi, en solo

Optimiseur de bytecode (peephole). Écris `optimiser(code)` qui supprime les calculs constants sur du code **sans saut** :

- `PUSH a; PUSH b; ADD` → `PUSH (a+b)` (idem SUB, MUL, et DIV si `b ≠ 0`, comparaisons) ;
- `PUSH a; NEG` → `PUSH -a` ;
- répète jusqu'à ce que plus rien ne change.

Pour un programme qui contient des sauts, ne touche à rien (les adresses cibles seraient décalées). Vérifie sur 200 programmes d'expressions aléatoires (`print` d'une expression avec nombres seulement) que la sortie est identique et que le code est plus court.

À toi, en solo · réponse

Pistes de correction

```
# (Compilateur, VM et OPS_BIN de l'exemple)
import random

def optimiser(code):
    if any(instr[0] in ("JMP", "JZ") for instr in code):
        return list(code)                          # on ne modifie pas un code avec
    sauts
    code = list(code)
    changement = True
    while changement:
        changement, i, nouveau = False, 0, []
        while i < len(code):
            if i + 2 < len(code) and code[i][0] == "PUSH" and code[i + 1][0] == "P
   USH" and code[i + 2][0] in OPS_BIN:
                a, b, op = code[i][1], code[i + 1][1], code[i + 2][0]
                if not (op == "DIV" and b == 0):
                    nouveau.append(("PUSH", OPS_BIN[op](a, b)))
                    i += 3
                    changement = True
```

À toi, en solo · réponse

Pistes de correction (suite)

```
                    continue
            if i + 1 < len(code) and code[i][0] == "PUSH" and code[i + 1][0] == "N
    EG":
                nouveau.append(("PUSH", -code[i][1]))
                i += 2
                changement = True
                continue
            nouveau.append(code[i])
            i += 1
        code = nouveau
    return code

def expression_aleatoire(rng, profondeur=3):
    if profondeur == 0 or rng.random() < 0.3:
        return str(rng.randint(1, 9))
    op = rng.choice(["+", "-", "*"])
    return f"({expression_aleatoire(rng, profondeur - 1)} {op} {expression_aleatoi
    re(rng, profondeur - 1)})"
```

À toi, en solo · réponse

Pistes de correction (suite)

```
rng = random.Random(5)
gagne = 0
for _ in range(200):
    source = f"print {expression_aleatoire(rng)};"
    code = compiler_source(source)
    opt = optimiser(code)
    assert executer_vm(opt) == executer_vm(code), source
    gagne += len(code) - len(opt)
print("instructions économisées :", gagne)

code = compiler_source("let i = 0; while i < 3 { i = i + 1; }")
assert optimiser(code) == code                    # avec sauts : inchangé
print(desassembler(optimiser(compiler_source("print 2 + 3 * 4;"))))
```

Le programme `print 2 + 3 * 4;` devient `PUSH 14; PRINT; HALT` : le calcul a été fait à la compilation (pliage de constantes, mission 4, appliqué au bytecode).

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Oublier le remplissage des sauts	Les `JZ` pointent sur `None`	Mémoriser l'indice du saut puis le compléter
Pile déséquilibrée	Résidus sur la pile ou dépilage d'une pile vide	Chaque expression pousse exactement une valeur ; une instruction expression la dépile (`POP`)
Optimiser du code avec des sauts	Les adresses de saut deviennent fausses	Recalculer les cibles ou limiter l'optimisation aux blocs sans saut
Comparer des vitesses sans vérifier l'égalité des résultats	Un VM rapide mais faux ne vaut rien	`assert` d'équivalence avant de mesurer

Mini-boss

Mini-boss

Transpileur HashScript → Python. Au lieu d'un bytecode, génère du **code source Python** à partir de l'AST (pour le sous-ensemble sans fonctions), puis exécute-le avec `exec`.

1. Écris `vers_python(programme)` : variables → variables Python, `print` → ajout à une liste de sortie (en respectant le format de HashScript), `si`/`tantque` → `if`/`while` avec indentation correcte.
2. Les erreurs de division par zéro doivent être converties en `ErreurLangage` (indice : une fonction auxiliaire `div(a, b)` générée dans le code).
3. Vérifie sur 20 programmes que l'interpréteur d'arbre, la VM et le transpileur donnent les mêmes sorties.
4. Mesure les trois sur une boucle de 100 000 itérations et classe-les.
5. Explique en 5 lignes quand préférer chaque approche (arbre, VM, transpilation).

Mini-boss · réponse

Pistes de correction

Un bon travail : une fonction récursive qui génère des lignes avec un niveau d'indentation (`"    " * niveau`), `let`/`Assign` → `nom = expression`, `Si` → `if cond:` + bloc (avec `pass` si vide) + `else:`, `TantQue` → `while cond:`, expressions entièrement parenthésées pour garder les priorités; `div` définie en tête du code généré et levant `ErreurLangage`; le code est exécuté avec `exec(code, {"sortie": [], "div": div})`. Résultats typiques : la transpilation (qui profite de la VM de Python) est la plus rapide, la VM maison est ensuite, l'interpréteur d'arbre le plus lent. Explication : arbre = simplicité et débogage ; VM = portabilité et contrôle (sécurité, limites) ; transpilation = performance maximale, mais dépend d'un langage cible et pose des questions de sécurité (code généré exécuté).

Quiz éclair

Quiz éclair

1. Un bytecode est :

A. du texte lisible par l'utilisateur
B. un arbre de syntaxe
C. une liste plate d'instructions simples exécutée par une VM

2. `PUSH 2; PUSH 3; ADD` laisse sur la pile :

A. 5
B. 2 et 3
C. rien

3. Le remplissage a posteriori (backpatching) sert à :

A. optimiser les boucles

Quiz éclair

Quiz éclair (suite)

B. compléter les cibles de saut une fois connues

C. annuler une instruction

4. 🔁 *Révision mission 07* : En portée lexicale, une variable libre d'une fonction est cherchée dans :

A. l'environnement de l'appelant

B. la mémoire globale uniquement

C. l'environnement où la fonction a été définie

5. 🔁 *Révision mission 07* : Une fermeture est :

A. une fonction accompagnée de son environnement de définition

B. une fonction sans paramètre

C. un type de boucle

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Quelle tâche de ta vie préfères-tu « traduire une fois en liste d'étapes simples » plutôt que de réfléchir à chaque fois ?
2. Quels compromis vois-tu entre un interpréteur d'arbre, une VM et un transpileur ?

À retenir

À retenir

1. Bytecode = liste plate d'instructions ; VM = boucle fetch-decode-execute.
2. Machine à pile : opérandes empilés, opérations sur le sommet.
3. Contrôle de flot : JMP, JZ, backpatching.
4. Désassembleur et tests d'équivalence.
5. Optimisation peephole sur du code sans saut.

MISSION ACCOMPLIE

+100 XP

Badge débloqué : Pilote de VM

Prochaine mission : Erreurs et diagnostics

