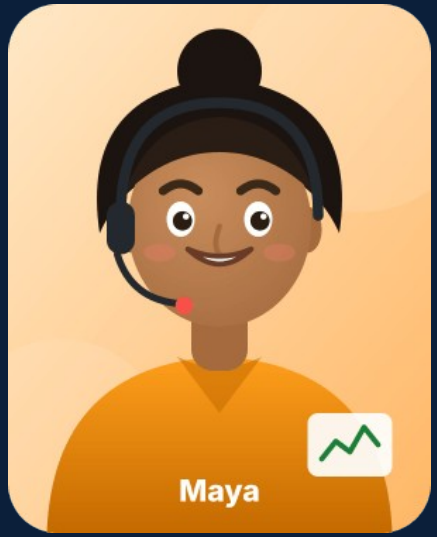
HEF-090 · MISSION 10 / 10

Un mini-langage de bout en bout

Lexer, parser, analyse, interpréteur, VM, diagnostics : tout se rejoint dans HashScript.



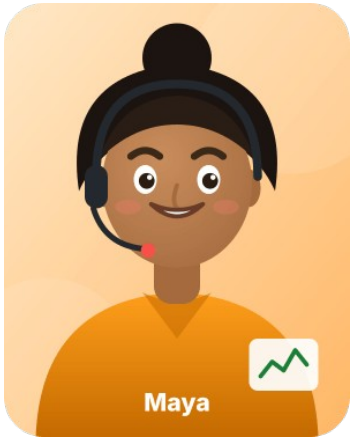


Badge : Créateur·rice de langage

110 min · intermédiaire

L'histoire

Dernier chapitre du module. Les formateurs de HashBox veulent utiliser HashScript pour écrire leurs exercices auto-corrigés. Maya demande une démonstration complète : un programme, ses erreurs éventuelles, son exécution.



> *« Tu as chaque brique, dit Sam. Il faut maintenant les assembler, tester l'ensemble comme un produit, et prouver qu'on peut le faire évoluer sans tout casser. »*

Tu assembles et fais évoluer HashScript, puis tu présentes l'ensemble comme une vraie chaîne de compilation.

Ta mission

À la fin, je serai capable de…

1. assembler lexer, parser, analyse sémantique, exécution et diagnostics en un pipeline
2. tester un langage avec une batterie de programmes de référence (tests de bout en bout)
3. écrire des programmes HashScript non triviaux
4. ajouter une fonctionnalité de langage de façon propre (grammaire, lexer, parser, sémantique, exécution, tests)
5. documenter un langage (grammaire, exemples, limites)

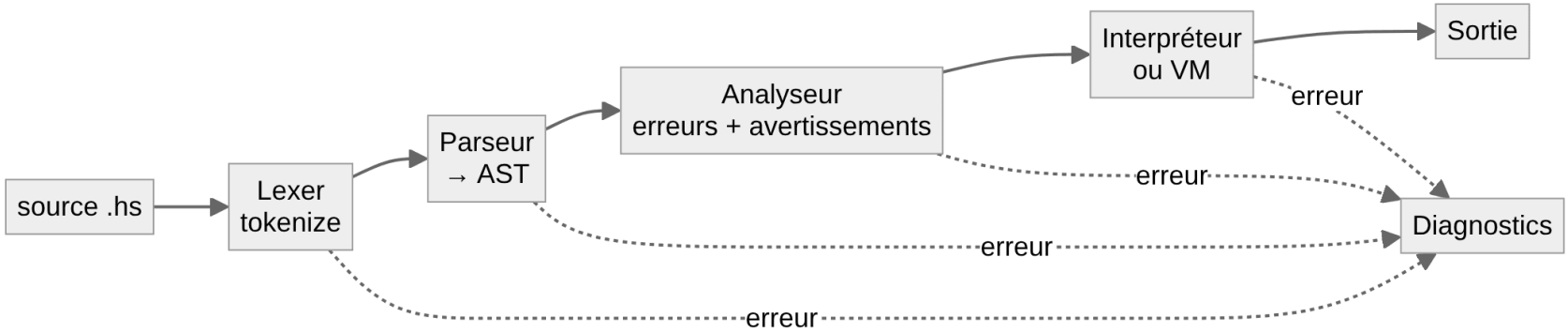
Défi d'échauffement

Défi d'échauffement

Tu veux ajouter l'opérateur reste de la division `%` à un langage existant. Fais la liste de **tous les endroits** où tu dois toucher (écriture du texte, découpage en mots, structure des phrases, vérification du sens, exécution). Dans quel ordre travaillerais-tu, et quels tests écrirais-tu **avant** de commencer ?

Le concept

Le pipeline complet



Description du schéma : le code source passe par le lexer, le parseur, l'analyseur puis l'interpréteur (ou la VM) pour produire la sortie. À chaque étape, une erreur est envoyée au module de diagnostics.

Le concept

Le pipeline complet (suite)

La version de référence de HashScript se trouve dans `01-Course/code/hashscript.py`, avec ses tests dans `test_hashscript.py`. Elle rassemble le travail des missions 2 à 9.

Les composants de la référence

Composant	Rôle
`tokenize(source)`	texte → jetons positionnés
`Parseur` / `parser(source)`	jetons → AST
`Analyseur` / `analyser(programme)`	erreurs et avertissements sémantiques
`Interprete`	exécution de l'AST (variables, fonctions, fermetures)
`executer_complet(source)`	pipeline complet → `(sortie, erreurs, avertissements)`
`ErreurLangage`	erreur positionnée, avec `afficher(source)`

Le concept

La grammaire de HashScript

```
programme    → instruction*
instruction  → "let" ID "=" expression ";"
             | "print" expression ";"
             | ID "=" expression ";"
             | "if" expression bloc ("else" bloc)?
             | "while" expression bloc
             | "fun" ID "(" paramètres? ")" bloc
             | "return" expression ";"
             | expression ";"
bloc         → "{" instruction* "}"
expression   → comparaison
comparaison  → somme (("==" | "!=" | "<" | ">" | "<=" | ">=") somme)*
somme        → produit (("+" | "-") produit)*
produit      → unaire (("*" | "/") unaire)*
unaire       → "-" unaire | appel
appel        → primaire ("(" arguments? ")")*
```

Le concept

La grammaire de HashScript (suite)

```
primaire     → NOMBRE | ID | "true" | "false" | "(" expression ")"
```

Faire évoluer un langage : la méthode

Pour ajouter une fonctionnalité, on suit toujours la même **checklist** :

1. **Spécifier** : écrire la grammaire et la sémantique, avec des exemples et des cas d'erreur.
2. **Écrire les tests d'abord** (bout en bout).
3. **Lexer** : de nouveaux jetons ?
4. **Parseur** : quelle(s) règle(s) ? Quelle priorité ?
5. **AST** : de nouveaux nœuds ?
6. **Analyse sémantique** : quelles vérifications (types, portées) ?
7. **Exécution** (interpréteur, et VM si elle existe).

Le concept

Faire évoluer un langage : la méthode (suite)

8. **Diagnostics** : messages et positions.
9. **Documentation** et mise à jour de la grammaire.

Étendre sans modifier (pour l'exercice)

Pour garder la référence intacte, on peut étendre par **héritage** et par **substitution contrôlée** de motifs. En pratique, on modifie directement les fichiers et on garde les tests au vert.

Tester un langage

Type de test	Exemple
Unitaire	lexer sur une chaîne, parseur sur une expression
De bout en bout	programme source → sortie attendue

Le concept

Tester un langage (suite)

Type de test	Exemple
Erreurs	source invalide → message, ligne et colonne
Propriétés	l'optimiseur ne change pas la sortie ; interpréteur = VM
Régression	chaque bug corrigé devient un test

🧠 ***Astuce*** *: un langage vit par ses tests. Plus ils sont complets, plus on ose le faire évoluer.*

Exemple pas à pas

Problème : exécuter une batterie de programmes de bout en bout, produire un rapport, et vérifier que l'interpréteur et la VM sont d'accord.

```
from hashscript import executer_complet, parser, executer

PROGRAMMES = {
    "factorielle": ("fun f(n) { if n <= 1 { return 1; } return n * f(n - 1); } pri
   nt f(10);", ["3628800"]),
    "fibonacci récursif": ("fun fib(n) { if n < 2 { return n; } return fib(n - 1)
   + fib(n - 2); } print fib(15);", ["610"]),
    "compteur (fermeture)": ("fun mk() { let c = 0; fun inc() { c = c + 1; return
   c; } return inc; } let a = mk(); a(); a(); print a();", ["3"]),
    "somme 1..100": ("let i = 1; let s = 0; while i <= 100 { s = s + i; i = i + 1;
    } print s;", ["5050"]),
    "pgcd (soustractions)": ("fun pgcd(a, b) { while a != b { if a > b { a = a - b
   ; } else { b = b - a; } } return a; } print pgcd(48, 18);", ["6"]),
    "puissance récursive": ("fun p(x, n) { if n == 0 { return 1; } return x * p(x,
    n - 1); } print p(2, 20);", ["1048576"]),
```

Exemple pas à pas (suite)

```
    "composition": ("fun add(n) { fun f(x) { return x + n; } return f; } fun twice
   (f) { fun g(x) { return f(f(x)); } return g; } print twice(add(3))(10);", ["16"
   ]),
    "portée de bloc": ("let x = 1; if true { let x = 2; print x; } print x;", ["2"
   , "1"]),
}

ok = 0
for nom, (source, attendu) in PROGRAMMES.items():
    sortie, erreurs, avertissements = executer_complet(source)
    statut = "OK " if sortie == attendu and not erreurs else "ÉCHEC"
    ok += statut.strip() == "OK"
    print(f"{statut} {nom:22s} {sortie}  {avertissements or ''}")
print(f"{ok}/{len(PROGRAMMES)} programmes corrects")

ERREURS = {
    "variable inconnue": "print z;",
    "division par zéro": "print 1 / 0;",
    "syntaxe": "let a = ;",
```

Exemple pas à pas (suite)

```
    "arité": "fun f(a) { return a; } print f(1, 2);",
}
for nom, source in ERREURS.items():
    _, erreurs, _ = executer_complet(source)
    ligne, colonne, message = erreurs[0]
    print(f"{nom:20s} → ligne {ligne}, colonne {colonne} : {message}")
```

Lecture : huit programmes de référence s'exécutent correctement (récursion, fermetures, boucles, composition de fonctions), et les quatre cas d'erreur produisent un message positionné. Cette batterie est **le filet de sécurité** qui permet de modifier le langage.

Faire passer HashScript en ligne de commande : le fichier de référence contient un point d'entrée :

```
python hashscript.py mon_programme.hs
```

Exemple pas à pas (suite)

Les avertissements et erreurs vont sur `stderr`, la sortie sur `stdout`, et le code de sortie du processus vaut 1 en cas d'erreur (voir HEF-050).

À toi, avec aide

Écris **trois programmes HashScript** de ton choix qui utilisent chacun une fonctionnalité différente (fonction récursive, fermeture, boucle avec condition). Pour chacun :

1. donne le programme et la sortie attendue ;
2. exécute-le avec `executer_complet` ;
3. introduis volontairement une faute (variable mal orthographiée, oubli de `;`, mauvais nombre d'arguments) et note le message obtenu.

Indices progressifs

Indice 1

Idées : puissance récursive, accumulateur par fermeture, nombre de doublements nécessaires pour dépasser 1000.

Indice 2

Un compteur ou un accumulateur par fermeture est un bon exemple de fonctionnalité à tester.

Indice 3

Pour la faute volontaire, change juste un caractère (un nom, un point-virgule, un argument).

À toi, avec aide · réponse

Réponse

```
from hashscript import executer_complet

programmes = [
    ("fun puissance(x, n) { if n == 0 { return 1; } return x * puissance(x, n - 1)
   ; } print puissance(3, 4);", ["81"]),
    ("fun accumulateur() { let total = 0; fun ajoute(n) { total = total + n; retur
   n total; } return ajoute; } let a = accumulateur(); a(5); a(7); print a(0);", [
   "12"]),
    ("let n = 1; let pas = 0; while n < 1000 { n = n * 2; pas = pas + 1; } print p
   as;", ["10"]),
]
for source, attendu in programmes:
    sortie, erreurs, _ = executer_complet(source)
    assert sortie == attendu and not erreurs, (source, sortie, erreurs)

fautes = [
    "fun puissance(x, n) { if n == 0 { return 1; } return x * puissanse(x, n - 1);
    } print puissance(3, 4);",
    "let n = 1 let pas = 0; print pas;",
```

À toi, avec aide · réponse

Réponse (suite)

```
    "fun accumulateur() { let total = 0; fun ajoute(n) { total = total + n; return
    total; } return ajoute; } let a = accumulateur(); print a(1, 2);",
]
for source in fautes:
    _, erreurs, _ = executer_complet(source)
    print(erreurs[0])
```

Les trois fautes produisent respectivement : « fonction inconnue 'puissanse' », « attendu ';' », « ajoute attend 1 argument(s), reçu 2 ».

À toi, en solo

Ajouter l'opérateur reste `%`. Suis la checklist pour ajouter `a % b` (reste de la division) à HashScript, avec la **priorité de `*` et `/`**, sans modifier le fichier de référence (on étend par héritage et substitution contrôlée du motif du lexer) :

1. tests d'abord : `print 17 % 5;` → `2`, `print 2 + 7 % 4 * 2;` → `8`, `print 5 % 0;` → erreur positionnée « modulo par zéro » ;
2. lexer : étendre le motif `OP` ;
3. parseur : sous-classe de `Parseur` dont `produit` accepte `%` ;
4. analyseur : `%` demande deux nombres ;
5. interpréteur : sous-classe de `Interprete` qui évalue `%` ;
6. un pipeline `executer_etendu(source)` qui assemble tout.

À toi, en solo · réponse

Pistes de correction

```
import re
import hashscript
from hashscript import Parseur, Interprete, Analyseur, Bin, ErreurLangage, tokeniz
   e

# 2. Lexer : on étend le motif OP et on recompile
hashscript.SPEC[3] = ("OP", r"==|!=|<=|>=|[-+*/%()<>=;{},]")
hashscript.MOTIF = re.compile("|".join(f"(?P<{n}>{p})" for n, p in hashscript.SPEC
   ))

# 3. Parseur : même priorité que * et /
class ParseurModulo(Parseur):
    def produit(self):
        g = self.unaire()
        while self.courant()[0] == "OP" and self.courant()[1] in ("*", "/", "%"):
            _, op, l, c = self.avancer()
            g = Bin(op, g, self.unaire(), l, c)
        return g
```

Pistes de correction (suite)

```
# 4. Analyse : % est arithmétique
class AnalyseurModulo(Analyseur):
    def type_de(self, n):
        if isinstance(n, Bin) and n.op == "%":
            g, d = self.type_de(n.gauche), self.type_de(n.droite)
            if "?" not in (g, d) and (g, d) != ("nombre", "nombre"):
                self.erreur(n, f"'%' attend deux nombres, reçu {g} et {d}")
            return "nombre"
        return super().type_de(n)

# 5. Exécution
class InterpreteModulo(Interprete):
    def eval(self, n, env):
        if isinstance(n, Bin) and n.op == "%":
            a, b = self.eval(n.gauche, env), self.eval(n.droite, env)
            if b == 0:
                raise ErreurLangage("modulo par zéro", n.ligne, n.col)
            return a % b
        return super().eval(n, env)
```

Pistes de correction (suite)

```
# 6. Pipeline étendu
def executer_etendu(source):
    try:
        programme = ParseurModulo(tokenize(source)).programme()
    except ErreurLangage as e:
        return [], [(e.ligne, e.colonne, e.message)], []
    erreurs, avertissements = AnalyseurModulo().analyser(programme)
    if erreurs:
        return [], erreurs, avertissements
    try:
        return InterpreteModulo().executer(programme), [], avertissements
    except ErreurLangage as e:
        return [], [(e.ligne, e.colonne, e.message)], avertissements

assert executer_etendu("print 17 % 5;")[0] == ["2"]
assert executer_etendu("print 2 + 7 % 4 * 2;")[0] == ["8"]
assert executer_etendu("print 10 % 3 + 10 / 4;")[0] == ["3.5"]
_, erreurs, _ = executer_etendu("print 5 % 0;")
```

Pistes de correction (suite)

```
assert erreurs and "modulo par zéro" in erreurs[0][2]
_, erreurs, _ = executer_etendu("print true % 2;")
assert erreurs and "attend deux nombres" in erreurs[0][2]
print("opérateur % ajouté sans toucher à la référence")
```

(En projet réel, tu modifierais directement `hashscript.py` à ces cinq endroits et tu ferais passer les tests.)

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Ajouter une fonctionnalité sans tests	On casse l'existant sans le voir	Tests de bout en bout d'abord
Oublier une phase	La fonction passe le parseur mais plante à l'exécution	Suivre la checklist : lexer, parseur, AST, sémantique, exécution, diagnostics
Changer la grammaire sans la documenter	Les utilisateurs découvrent le comportement par accident	Mettre la grammaire à jour avec le code
Un seul mode d'exécution testé	Interpréteur et VM divergent	Test de propriété : mêmes sorties

Mini-boss

Mini-boss

Projet final du module HEF-090 : étendre HashScript. Choisis **deux** extensions et implémente-les proprement à travers tout le pipeline, avec grammaire mise à jour, tests de bout en bout (au moins 8 par extension), messages d'erreur soignés et une note de conception :

- **Booléens logiques** : `et`, `ou`, `non` (court-circuit pour `et`/`ou`) ;
- **Chaînes de caractères** : littéraux `"…"`, concaténation avec `+`, longueur, comparaison ;
- **Boucle `pour`** : `pour i de 1 a 10 { … }` ;
- **Listes** : littéraux `[1, 2, 3]`, indexation `l[0]`, fonction native `longueur`, ajout ;
- **Commentaires de bloc** et **constantes** (`const`).

Livrables :

1. le code modifié (lexer, parseur, analyseur, interpréteur) avec tous les tests au vert ;
2. la grammaire EBNF mise à jour ;

Mini-boss

Mini-boss (suite)

3. un ADR pour chaque extension (alternatives envisagées, conséquences) ;
4. si tu as fait la VM de la mission 8, son extension aussi (ou une justification de ne pas la faire) ;
5. un petit programme de démonstration qui utilise tes deux extensions ;
6. une présentation de 10 lignes : ce que tu as appris sur la conception d'un langage.

Mini-boss · réponse

Pistes de correction

Pistes. **Booléens logiques** : mots-clés `et`, `ou`, `non` dans `MOTS_CLES` ; deux niveaux de grammaire entre `expression` et `comparaison` (`ou → et ("ou" et)*`, `et → non ("et" non)*`, `non → "non" non | comparaison`) ; évaluation avec court-circuit (ne pas évaluer la droite si la gauche suffit) ; l'analyseur exige des booléens. **Chaînes** : motif `CHAINE` dans le lexer, nœud `Chaine(valeur)`, `+` surchargé (nombre+nombre ou chaîne+chaîne, erreur sinon), `formater` sans guillemets. **Pour** : sucre syntaxique : le parseur peut produire un bloc `Let` + `TantQue` équivalent (désucrage), sans toucher à l'interpréteur. **Listes** : nœuds `Liste` et `Indice`, vérification des bornes avec message positionné, affectation indexée optionnelle. Un bon dossier montre le **désucrage** comme technique d'ajout peu coûteux, des tests de non-régression sur les 8 programmes de référence, et un ADR qui compare au moins deux alternatives (par exemple court-circuit ou évaluation stricte).

Quiz éclair

Quiz éclair

1. Pour ajouter un opérateur à un langage, il faut modifier :

A. lexer, parseur, analyse, exécution et tests
B. seulement le lexer
C. seulement l'interpréteur

2. Un test de bout en bout d'un langage vérifie :

A. uniquement le lexer
B. qu'un programme source donne la sortie attendue
C. la vitesse du processeur

3. Le désucrage consiste à :

A. supprimer les commentaires

Quiz éclair

Quiz éclair (suite)

B. compresser le programme
C. traduire une syntaxe pratique en constructions plus simples du langage

4. 🔁 *Révision mission 09* : Un bon message d'erreur indique :

A. où, quoi, pourquoi et comment corriger
B. seulement « erreur »
C. un code interne

5. 🔁 *Révision mission 09* : La récupération d'erreur du parseur consiste à :

A. s'arrêter à la première erreur
B. avancer jusqu'à un point de synchronisation puis reprendre
C. ignorer toutes les erreurs

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Quel élément de la chaîne de compilation t'a le plus surpris·e en profondeur, et lequel te semble le plus utile dans ton travail quotidien ?
2. Quel petit langage (règles, requêtes, configuration) pourrais-tu concevoir pour un outil que tu utilises ?

À retenir

À retenir

1. Pipeline : lexer → parseur → analyse → exécution, avec diagnostics à chaque étape.
2. Ajouter une fonctionnalité : spécifier, tester, lexer, parseur, AST, sémantique, exécution, diagnostics, doc.
3. Tests de bout en bout, d'erreurs, de propriétés, de régression.
4. Le désucrage ajoute de la syntaxe sans toucher à l'exécution.
5. Un langage vit par ses tests et sa documentation.

Félicitations

Tu as terminé **HEF-090 — Compilateurs et langages**. Tu sais décrire un langage par une grammaire, écrire un lexer et un parseur par descente récursive, manipuler un AST, vérifier la sémantique, exécuter un programme par interpréteur ou par machine virtuelle, soigner les diagnostics, et faire évoluer un langage proprement.
→ Prochain module : **HEF-100 — Sécurité**

MISSION ACCOMPLIE

+120 XP

Badge débloqué : Créateur·rice de langage

Fin du module HEF-090. Bravo !

