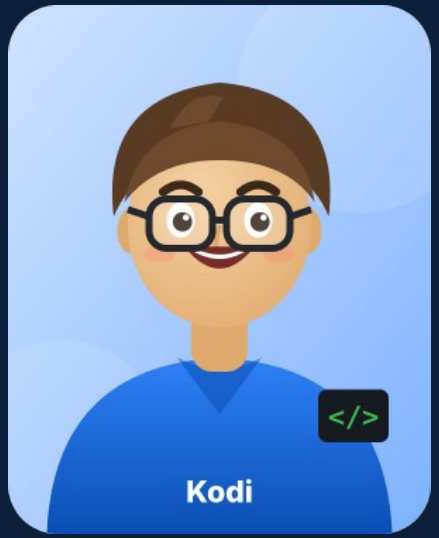
HEF-090 · MISSION 09 / 10

Erreurs et diagnostics

Un bon compilateur ne se contente pas de refuser : il explique, localise et suggère.



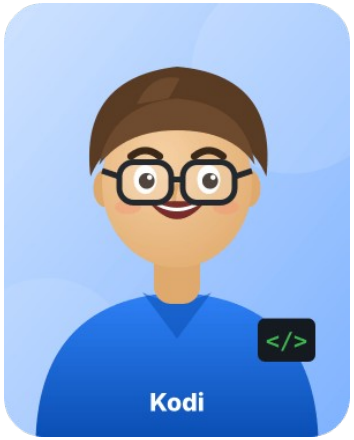


Badge : Médecin des messages

85 min · intermédiaire

L'histoire

Les formateurs de HashBox testent HashScript. Beaucoup abandonnent après la première erreur : `erreur ligne 7` sans autre détail. Un formateur écrit `compter` à un endroit et `compteur` à un autre : « variable inconnue », sans plus.



> *« Les messages d'erreur sont l'interface principale d'un langage pour un débutant, dit Maya. Il faut montrer la ligne, pointer l'endroit exact, et si possible proposer la correction. »*

Tu améliores le diagnostic de HashScript : positions, extraits, suggestions, plusieurs erreurs à la fois.

Ta mission

À la fin, je serai capable de…

1. afficher un extrait de code avec un pointeur ^ sous l'endroit fautif
2. formuler des messages utiles : quoi, où, pourquoi, comment corriger
3. suggérer une correction par proximité de noms (« vouliez-vous dire … ? »)
4. récupérer après une erreur de syntaxe pour en signaler plusieurs
5. distinguer erreurs, avertissements et notes

Défi d'échauffement

Compare deux retours d'un correcteur : (A) « Faux. » (B) « Ligne 3, mot 4 : "mangés" devrait être "mangé" car le sujet est au singulier ("le chat"). » Lequel te permet de corriger ? Liste les **quatre éléments** qui rendent B utile (où, quoi, pourquoi, comment).

Le concept

Les qualités d'un bon message d'erreur

Élément	Question	Exemple
Où ?	ligne, colonne, extrait	`ligne 3, colonne 9` + la ligne de code
Quoi ?	nature du problème	« variable inconnue `compter` »
Pourquoi ?	cause	« aucune déclaration `let compter` dans cette portée »
Comment ?	piste de correction	« vouliez-vous dire `compteur` ? »

Un bon message est **court**, **précis**, **dans le vocabulaire de l'utilisateur** (pas celui de l'implémentation : jamais « jeton inattendu de type OP » sans traduire).

Le concept

Pointer l'endroit exact

```
erreur ligne 2, colonne 9 : variable inconnue 'compter'
  print compter + 1;
        ^^^^^^^
```

On affiche la ligne source et des `^` sous la zone fautive (colonne de début, longueur du jeton).

Suggérer : la distance d'édition

Pour « vouliez-vous dire … ? », on compare le nom inconnu aux noms **connus** avec une mesure de **similarité**. En Python : `difflib.get_close_matches(mot, candidats)` (basé sur la plus longue sous-séquence commune). La distance de **Levenshtein** (nombre minimal d'insertions, suppressions, substitutions) en est une cousine.

Le concept

Plusieurs erreurs d'un coup : la récupération

Si le parser s'arrête à la première erreur, l'utilisateur corrige, relance, découvre la suivante… épuisant. La **récupération d'erreur** consiste à, après une erreur, **avancer jusqu'à un point de synchronisation** (la fin de l'instruction `;`, ou un mot-clé qui commence une instruction) et à **reprendre** l'analyse.

```
class ParseurTolerant(Parseur):
    def programme(self):
        instrs = []
        while self.courant()[0] != "FIN":
            debut = self.i
            try:
                instrs.append(self.instruction())
            except ErreurLangage as e:
                self.erreurs.append(e)
                self.synchroniser()
                if self.i == debut:
```

Le concept

Plusieurs erreurs d'un coup : la récupération (suite)

```
                self.avancer()           # garantit qu'on progresse toujours
        return instrs
```

Piège : sans la garantie de progression, une erreur sur le premier jeton fait boucler à l'infini.

Éviter les erreurs en cascade

Une erreur de syntaxe peut en produire dix autres (le parseur est « perdu »). Bonnes pratiques :

- ne signaler qu'une erreur **par instruction** ;
- après une erreur, ne pas analyser sémantiquement l'instruction invalide ;
- limiter le nombre d'erreurs affichées (par exemple 10).

Le concept

Niveaux de gravité

Niveau	Effet
Erreur	le programme est rejeté
Avertissement	le programme passe, mais c'est suspect
Note / aide	information complémentaire attachée à une erreur

Codes d'erreur et documentation

Les langages sérieux attribuent un **code** à chaque erreur (`E0425` en Rust) et une page d'explication avec exemples. Cela permet de chercher, de documenter et de tester les messages.

Le concept

Tester les messages

Les messages font partie du **contrat** du langage : on écrit des tests qui vérifient le texte, la position et l'extrait, comme pour n'importe quelle sortie.

🧠 ***Astuce** : l'erreur la plus utile est celle qui vient avec sa correction. Si tu sais ce que l'utilisateur voulait probablement faire, dis-le.*

Exemple pas à pas

Problème : un rapport d'erreurs complet pour HashScript, avec extraits, suggestions et récupération.

```
import difflib
from hashscript import Parseur, tokenize, ErreurLangage, Var, Let

class ParseurTolerant(Parseur):
    def __init__(self, jetons):
        super().__init__(jetons)
        self.erreurs = []

    def synchroniser(self):
        while self.courant()[0] != "FIN":
            if self.courant()[1] == ";":
                self.avancer()
                return
            if self.courant()[0] == "MOT" and self.courant()[1] in ("let", "print"
   , "if", "while", "fun", "return"):
                return
```

Exemple pas à pas (suite)

```
            self.avancer()

    def programme(self):
        instrs = []
        while self.courant()[0] != "FIN":
            debut = self.i
            try:
                instrs.append(self.instruction())
            except ErreurLangage as e:
                self.erreurs.append(e)
                self.synchroniser()
                if self.i == debut:
                    self.avancer()
        return instrs

def rapport(source, erreurs):
    lignes = source.split("\n")
    morceaux = []
    for e in erreurs:
```

Exemple pas à pas

Exemple pas à pas (suite)

```
        texte = lignes[e.ligne - 1] if 1 <= e.ligne <= len(lignes) else ""
        suite = texte[e.colonne - 1:]
        longueur = max(1, len(suite.split()[0]) if suite.split() else 1)
        morceaux.append(f"erreur ligne {e.ligne}, colonne {e.colonne} : {e.message
   }\n  {texte}\n  {' ' * (e.colonne - 1)}{'^' * longueur}")
    return "\n\n".join(morceaux)

def variables_declarees(instrs):
    return {i.nom for i in instrs if isinstance(i, Let)}

def suggerer(nom, connus):
    proches = difflib.get_close_matches(nom, sorted(connus), n=1, cutoff=0.6)
    return f"vouliez-vous dire {proches[0]!r} ?" if proches else None

source = '''let compteur = 0;
let x = ;
print (1 + 2;
let = 5;
print compteur + 1;'''
```

Exemple pas à pas (suite)

```
parseur = ParseurTolerant(tokenize(source))
programme = parseur.programme()
print(f"{len(parseur.erreurs)} erreurs de syntaxe, {len(programme)} instructions v
   alides\n")
print(rapport(source, parseur.erreurs))

print()
print(suggerer("compter", variables_declarees(programme)))        # vouliez-vous d
   ire 'compteur' ?
print(suggerer("zzz", variables_declarees(programme)))            # None
```

Résultat attendu : trois erreurs de syntaxe signalées **en une seule passe** (`let x = ;`, `print (1 + 2;`, `let = 5;`), chacune avec son extrait et un pointeur ; la ligne valide `let compteur = 0;` et `print compteur + 1;` sont analysées normalement. La suggestion « vouliez-vous dire `compteur` ? » est calculée par similarité.

Lecture :

Exemple pas à pas (suite)

- `synchroniser` saute jusqu'au `;` ou à un mot-clé d'instruction ;
- la vérification `self.i == debut` évite la boucle infinie ;
- le pointeur `^` est aligné sur la colonne exacte de l'erreur.

À toi, avec aide

Étends `rapport` pour que chaque erreur affiche aussi une **ligne d'aide** (`aide :`) selon son type, en détectant des motifs dans le message :

- message contenant « attendu ';' » → « as-tu oublié le point-virgule à la fin de l'instruction ? » ;
- message contenant « expression attendue » → « il manque une valeur ou une variable ici » ;
- message contenant « accolade fermante » → « chaque '{' doit avoir son '}' ».

Teste avec trois sources invalides.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Utilise un dictionnaire `motif → aide` et teste `motif in message`.

Indice 2

Place l'aide après le pointeur, indentée.

Indice 3

Pour `print 1` sans `;`, le message du parseur est du type `attendu ';', trouvé …`.

À toi, avec aide · réponse

Réponse

```
# (ParseurTolerant et rapport de l'exemple)
AIDES = {
    "attendu ';'": "as-tu oublié le point-virgule à la fin de l'instruction ?",
    "expression attendue": "il manque une valeur ou une variable ici.",
    "accolade fermante": "chaque '{' doit avoir son '}'.",
}

def rapport_avec_aide(source, erreurs):
    base = rapport(source, erreurs).split("\n\n")
    morceaux = []
    for e, texte in zip(erreurs, base):
        aide = next((a for motif, a in AIDES.items() if motif in e.message), None)
        morceaux.append(texte + (f"\n  aide : {aide}" if aide else ""))
    return "\n\n".join(morceaux)

for source in ("print 1", "let a = 1 +;", "if true { print 1;"):
    p = ParseurTolerant(tokenize(source))
    p.programme()
    print(rapport_avec_aide(source, p.erreurs))
```

Réponse (suite)

```
    print("---")
```

Une table de messages d'aide est facile à enrichir et à tester.

À toi, en solo

Banc de tests des messages. Les messages font partie du contrat du langage. Écris un petit cadre de tests où chaque cas est `(source, [(ligne, colonne, fragment_du_message), …])` :

1. au moins 12 cas (syntaxe, caractère inconnu, variable inconnue avec suggestion, division par zéro, mauvais nombre d'arguments, accolade manquante) ;
2. le cadre exécute le pipeline complet (`executer_complet` de la référence ou ton propre assemblage) et compare erreurs trouvées et attendues ;
3. un rapport clair des cas qui échouent ;
4. ajoute une vérification : **aucune** erreur ne doit avoir une position `(0, 0)` sauf les erreurs globales explicitement autorisées.

À toi, en solo · réponse

Pistes de correction

```
from hashscript import executer_complet

CAS = [
    ("print x;", [(1, 7, "variable inconnue")]),
    ("let y = 3 +;", [(1, 12, "expression attendue")]),
    ("print 1 / 0;", [(1, 9, "division par zéro")]),
    ("let a = 1;\nprint a $ 2;", [(2, 9, "caractère inattendu")]),
    ("if true { print 1;", [(1, 19, "accolade")]),
    ("fun f(a) { return a; } print f(1, 2);", [(1, 31, "attend 1 argument")]),
    ("let a = 1; let a = 2;", [(1, 12, "déjà déclarée")]),
    ("let a = true + 1;", [(1, 14, "attend deux nombres")]),
    ("if 1 { print 1; }", [(1, 4, "condition")]),
    ("let a = 1; a = true;", [(1, 12, "de type")]),
    ("return 3;", [(0, 0, "return en dehors")]),
    ("print 1 +;", [(1, 10, "expression attendue")]),
]
GLOBALES_OK = {"return en dehors"}

echecs = []
```

À toi, en solo · réponse

Pistes de correction (suite)

```
for source, attendues in CAS:
    _, erreurs, _ = executer_complet(source)
    trouvees = [(l, c, m) for l, c, m in erreurs]
    for (l, c, fragment) in attendues:
        if not any(tl == l and tc == c and fragment in tm for tl, tc, tm in trouve
   es):
            echecs.append((source, attendues, trouvees))
            break
    for l, c, m in trouvees:
        if (l, c) == (0, 0) and not any(g in m for g in GLOBALES_OK):
            echecs.append((source, "position (0, 0) non autorisée", trouvees))

print(len(CAS), "cas,", len(echecs), "échec(s)")
for e in echecs:
    print(e)
```

Un cadre de ce type fige les messages : si un message change, un test échoue, et on décide en connaissance de cause.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Messages vagues (« syntaxe invalide »)	L'utilisateur ne sait pas quoi corriger	Où, quoi, pourquoi, comment
Boucle infinie dans la récupération	Le parseur ne progresse pas après une erreur	Garantir l'avancement d'au moins un jeton
Erreurs en cascade	Dix messages pour une faute	Une erreur par instruction, limite d'affichage
Vocabulaire interne	« jeton OP inattendu » incompréhensible	Parler le langage de l'utilisateur

Mini-boss

Mini-boss

Le meilleur message d'erreur de HashScript. Construis un module `diagnostic.py` qui, pour un programme, produit un rapport complet :

1. erreurs lexicales, syntaxiques (avec récupération, jusqu'à 10 erreurs), sémantiques ;
2. extrait de code, pointeur `^` de la bonne longueur, code d'erreur (`E001`…), aide ;
3. suggestions « vouliez-vous dire … ? » pour variables, fonctions et mots-clés mal orthographiés (`prnt` → `print`, `whlie` → `while`) ;
4. avertissements (variables inutilisées, code inatteignable) clairement séparés ;
5. un résumé final (« 3 erreurs, 2 avertissements »).

Teste avec 15 programmes, y compris un programme avec 5 fautes simultanées. Mesure la **qualité** : pour chaque programme, vérifie que corriger les erreurs dans l'ordre indiqué conduit à un programme sans erreur (simulation de correction automatique pour 5 cas simples).

Mini-boss · réponse

Pistes de correction

Un bon travail : table `CODES = {"variable inconnue": ("E010", aide), ...}` ; pour les mots-clés mal orthographiés, une erreur lexicale n'existe pas (c'est un identifiant valide `prnt`), donc le diagnostic repère un ID en début d'instruction suivi d'un token qui n'est pas `=`, `(`, ou `;` et propose le mot-clé le plus proche; la sortie est construite par une fonction `formater(erreur, source)`; la simulation de correction applique la suggestion textuelle (remplacement du nom) et relance le pipeline pour vérifier que l'erreur disparaît; le résumé compte erreurs et avertissements. Les tests comparent les rapports complets à des textes de référence (tests « golden »).

Quiz éclair

1. Un bon message d'erreur indique :

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

2. La récupération d'erreur du parseur consiste à :

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

3. `difflib.get_close_matches` sert ici à :

A. proposer un nom proche de celui qui est inconnu

Quiz éclair

Quiz éclair (suite)

B. trier les erreurs
C. chiffrer les messages

4. 🔁 *Révision mission 08* : Un bytecode est :

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

5. 🔁 *Révision mission 08* : Le remplissage a posteriori (backpatching) sert à :

A. optimiser les boucles
B. annuler une instruction
C. compléter les cibles de saut une fois connues

Quiz éclair · réponse

Réponses

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

Je réfléchis

Je réfléchis

1. Quel message d'erreur t'a le plus aidé, et lequel le plus frustré(e), dans un outil que tu utilises ?
2. Comment reformulerais-tu une erreur technique pour un utilisateur non technique ?

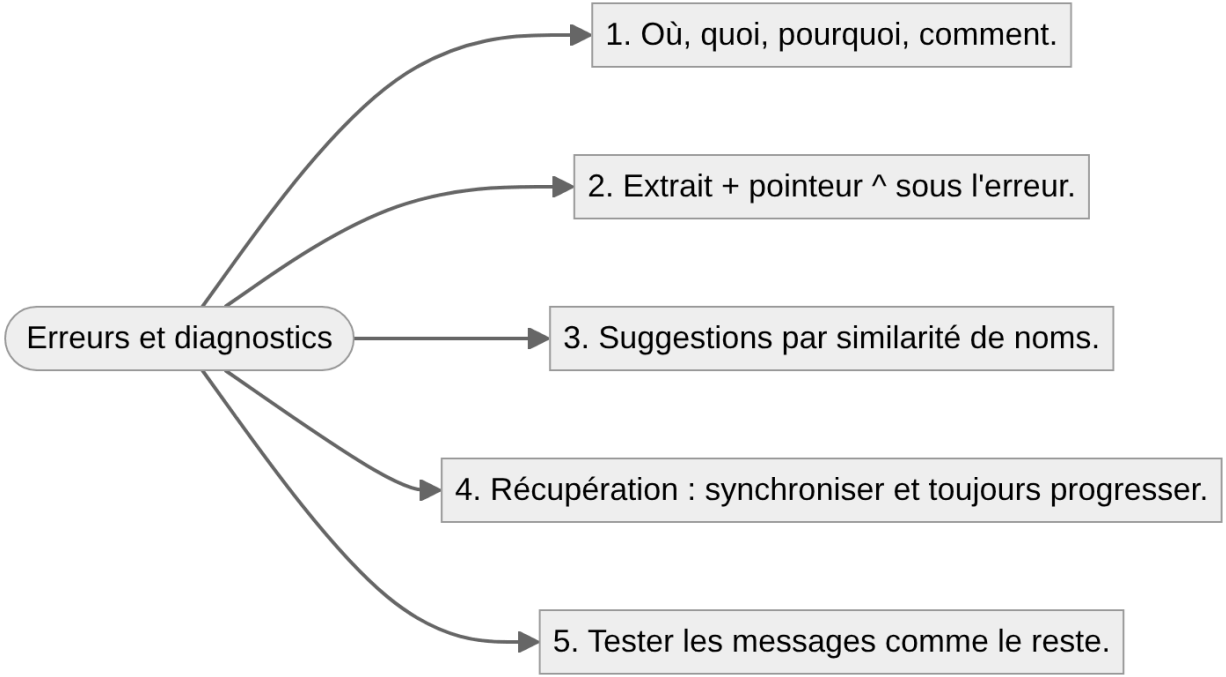
À retenir

À retenir

1. Où, quoi, pourquoi, comment.
2. Extrait + pointeur ^ sous l'erreur.
3. Suggestions par similarité de noms.
4. Récupération : synchroniser et toujours progresser.
5. Tester les messages comme le reste.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Médecin des messages

Prochaine mission : Un mini-langage de bout en bout

