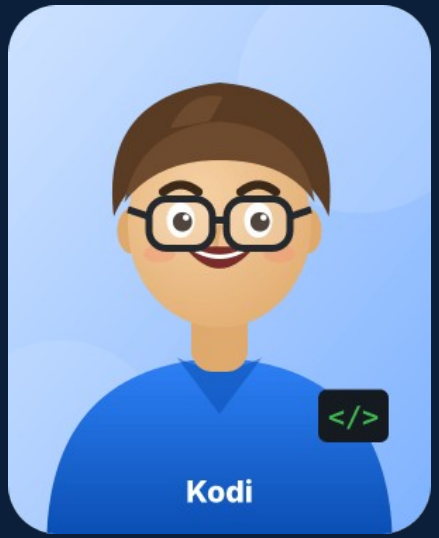
HEF-090 · MISSION 05 / 10

L'analyse sémantique



Un programme peut être bien écrit et pourtant absurde. L'analyse sémantique attrape ces erreurs avant l'exécution.



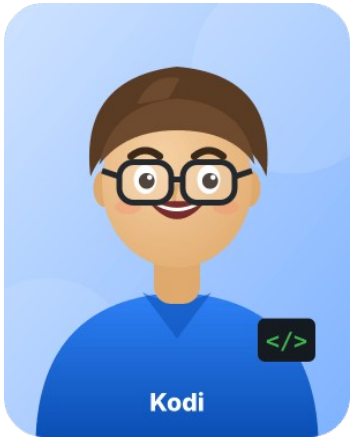
Badge : Inspecteur de sens

90 min · intermédiaire

L'histoire

Un formateur de HashBox écrit : `print compteur;` sans avoir jamais déclaré `compteur`, puis `let x = true + 1;`. Le parser dit « syntaxe correcte ». Le programme plante à l'exécution, au milieu d'une démonstration.

> *« La syntaxe dit si la phrase est bien construite, dit Sam. Pas si elle a du sens. Une deuxième passe doit vérifier : les noms existent-ils ? Les types sont-ils compatibles ? »*



Tu écris l'analyseur sémantique de HashScript : table des symboles, portées, vérification de types, avertissements.

Ta mission

À la fin, je serai capable de…

1. expliquer ce que vérifie l'analyse sémantique
2. construire une table des symboles avec des portées imbriquées
3. détecter variables non déclarées, redéclarations, types incompatibles
4. produire erreurs et avertissements positionnés
5. distinguer typage statique et dynamique

Défi d'échauffement

Défi d'échauffement

Voici trois phrases françaises syntaxiquement correctes : « Le triangle dort sous la racine carrée. », « Marie a mangé son sandwich. », « Il l'a donnée à elle. » Laquelle est **bien formée et sensée** ? Pour la dernière, que faut-il savoir pour la comprendre ? Transpose en programmation : `x + 1` est-elle toujours correcte ? Que dois-tu connaître de `x` ?

Le concept

Ce que vérifie l'analyse sémantique

Après le parsing, on dispose d'un AST **bien formé**. L'analyse sémantique contrôle les règles qui ne peuvent pas s'écrire dans une grammaire hors-contexte :

Contrôle	Exemple d'erreur
Résolution de noms	utiliser une variable non déclarée
Déclarations	déclarer deux fois le même nom dans la même portée
Types	additionner un booléen et un nombre
Appels	appeler une fonction avec le mauvais nombre d'arguments
Contrôle de flot	`return` hors d'une fonction ; code inatteignable
Initialisation	lire une variable avant de lui avoir donné une valeur

Le concept

La table des symboles

La **table des symboles** associe chaque nom à ses informations (type, portée, position de déclaration, utilisée ou non). On l'organise en **portées imbriquées** : une pile de dictionnaires, un par bloc.

```
portée globale   : {x: nombre, f: fonction}
  └─ bloc if     : {y: booléen}
       └─ bloc   : {x: booléen}     ← masque (shadowing) le x global
```

- **Déclaration** : ajoute un nom dans la portée **courante**.
- **Recherche** : du plus interne vers le plus externe (portée lexicale).
- **Sortie d'un bloc** : on dépile ; c'est le moment de détecter les variables jamais utilisées (avertissement).

Le concept

Types

Typage statique : les types sont vérifiés **avant** l'exécution (Rust, Java, TypeScript). **Typage dynamique** : vérifiés **pendant** (Python, JavaScript). Un analyseur peut ajouter un contrôle statique à un langage dynamique (comme `mypy` pour Python).

Pour HashScript, on distingue deux types simples : `nombre` et `booleen`, plus `?` (inconnu, pour les fonctions que l'on ne suit pas).

Opération	Opérandes	Résultat
`+ - * /`	nombre, nombre	nombre
`< > <= >=`	nombre, nombre	booleen
`== !=`	même type des deux côtés	booleen

Le concept

Types (suite)

Opération	Opérandes	Résultat
- unaire	nombre	nombre
condition `if` / `while`	booleen	:

L'analyse sémantique peut être **plus stricte** que l'exécution : l'interpréteur accepterait `if 1 { … }`, l'analyseur le signale quand même (c'est un choix de conception du langage).

Erreurs et avertissements

- **Erreur** : le programme est invalide, on ne l'exécute pas.
- **Avertissement** : le programme est valide mais suspect (variable jamais lue, code mort).

Un bon analyseur ne s'arrête pas à la première erreur : il **continue** pour en signaler plusieurs (en considérant le type d'une expression erronée comme `?` pour éviter les erreurs en cascade).

Le concept

Un analyseur en Python

```
from dataclasses import dataclass

@dataclass
class Info:
    type: str
    utilisee: bool
    ligne: int
    col: int

class Analyseur:
    def __init__(self):
        self.portees = [{}]
        self.erreurs, self.avertissements = [], []

    def ouvrir(self):
        self.portees.append({})
```

Le concept

Un analyseur en Python (suite)

```
    def fermer(self):
        for nom, info in self.portees.pop().items():
            if not info.utilisee:
                self.avertissements.append((info.ligne, info.col, f"variable {nom!
r} jamais utilisée"))

    def declarer(self, nom, type_, ligne, col):
        if nom in self.portees[-1]:
            self.erreurs.append((ligne, col, f"{nom!r} est déjà déclarée dans cett
e portée"))
        self.portees[-1][nom] = Info(type_, False, ligne, col)

    def chercher(self, nom):
        for portee in reversed(self.portees):
            if nom in portee:
                return portee[nom]
        return None
```

Le concept

Un analyseur en Python (suite)

🧠 ***Astuce*** *: l'analyse sémantique est ce qui fait que les messages d'erreur de ton langage semblent « intelligents ». Les meilleurs compilateurs y investissent beaucoup.*

Exemple pas à pas

Problème : analyser des programmes HashScript et rapporter erreurs et avertissements positionnés.

```
from dataclasses import dataclass
from hashscript import parser, Let, Assign, Print, Si, TantQue, Num, Bool, Var, Bi
   n, Neg, Fun, Appel, Retour

@dataclass
class Info:
    type: str
    utilisee: bool
    ligne: int
    col: int

class Analyseur:
    def __init__(self):
        self.portees = [{}]
        self.erreurs, self.avertissements = [], []
```

Exemple pas à pas (suite)

```
    def ouvrir(self): self.portees.append({})
    def fermer(self):
        for nom, info in self.portees.pop().items():
            if not info.utilisee:
                self.avertissements.append((info.ligne, info.col, f"variable {nom!
r} jamais utilisée"))

    def chercher(self, nom):
        for p in reversed(self.portees):
            if nom in p: return p[nom]

    def erreur(self, n, message):
        self.erreurs.append((getattr(n, "ligne", 0), getattr(n, "col", 0), message
))

    def type_de(self, n):
        if isinstance(n, Num): return "nombre"
        if isinstance(n, Bool): return "booleen"
        if isinstance(n, Var):
```

Exemple pas à pas

Exemple pas à pas (suite)

```
            info = self.chercher(n.nom)
            if info is None:
                self.erreur(n, f"variable inconnue {n.nom!r}")
                return "?"
            info.utilisee = True
            return info.type
        if isinstance(n, Neg):
            t = self.type_de(n.expr)
            if t not in ("nombre", "?"): self.erreur(n, "le moins unaire s'appliqu
    e à un nombre")
            return "nombre"
        if isinstance(n, Bin):
            g, d = self.type_de(n.gauche), self.type_de(n.droite)
            if "?" in (g, d):
                return "booleen" if n.op in ("<", ">", "<=", ">=", "==", "!=") els
    e "?" if n.op not in "+-*/" else "nombre"
            if n.op in "+-*/":
                if (g, d) != ("nombre", "nombre"): self.erreur(n, f"'{n.op}' atten
    d deux nombres, reçu {g} et {d}")
```

Exemple pas à pas (suite)

```
            return "nombre"
        if n.op in ("<", ">", "<=", ">="):
            if (g, d) != ("nombre", "nombre"): self.erreur(n, f"'{n.op}' atten
d deux nombres, reçu {g} et {d}")
            return "booleen"
        if g != d: self.erreur(n, f"comparaison entre types différents : {g} e
t {d}")
        return "booleen"
    return "?"                                          # appels, etc. : non su
ivis ici

 def instr(self, n):
    if isinstance(n, Let):
        t = self.type_de(n.expr)
        if n.nom in self.portees[-1]:
            self.erreur(n, f"{n.nom!r} est déjà déclarée dans cette portée")
        self.portees[-1][n.nom] = Info(t, False, n.ligne, n.col)
    elif isinstance(n, Assign):
        info = self.chercher(n.nom)
```

Exemple pas à pas

Exemple pas à pas (suite)

```
            t = self.type_de(n.expr)
            if info is None: self.erreur(n, f"affectation à une variable non décla
    rée {n.nom!r}")
            elif "?" not in (t, info.type) and t != info.type:
                self.erreur(n, f"{n.nom!r} est de type {info.type}, valeur de type
     {t}")
        elif isinstance(n, Print):
            self.type_de(n.expr)
        elif isinstance(n, (Si, TantQue)):
            t = self.type_de(n.cond)
            if t not in ("booleen", "?"): self.erreur(n.cond, f"la condition doit
    être booléenne, pas {t}")
            for bloc in ([n.alors, n.sinon] if isinstance(n, Si) else [n.corps]):
                self.ouvrir()
                for i in bloc: self.instr(i)
                self.fermer()

     def analyser(self, programme):
        for i in programme: self.instr(i)
```

Exemple pas à pas

Exemple pas à pas (suite)

```
        self.fermer()
        return sorted(self.erreurs), sorted(self.avertissements)

source = '''let a = 1;
let a = 2;
print b;
let c = true + 1;
if a { print 1; }
let inutile = 5;
let ok = a < 3;
while ok { ok = false; }
let d = 1;
d = true;
'''
erreurs, avertissements = Analyseur().analyser(parser(source))
for l, c, m in erreurs: print(f"erreur ligne {l}, col {c} : {m}")
for l, c, m in avertissements: print(f"avertissement ligne {l}, col {c} : {m}")
```

Exemple pas à pas (suite)

Lecture : l'analyseur signale plusieurs problèmes d'un coup : redéclaration, variable inconnue, types incompatibles, condition non booléenne, affectation de type incorrect. Il signale aussi `inutile` comme jamais utilisée. Le type `?` évite des erreurs en cascade quand une sous-expression est déjà fausse.

À toi, avec aide

Ajoute à l'analyseur une règle : **une variable utilisée avant d'avoir reçu une valeur** n'existe pas ici (toutes les variables sont initialisées à la déclaration), mais on veut signaler les **variables déclarées dans un bloc et utilisées après ce bloc**.

Écris deux programmes de test :

```
if true { let x = 1; }
print x;
```

et un programme valide où x est déclaré avant le bloc. Vérifie que le premier produit « variable inconnue 'x' » et le second aucune erreur.

Indices progressifs

Indice 1

La sortie du bloc dépile la portée : `x` n'existe plus après.

Indice 2

La fermeture de portée est déjà gérée par `fermer()` dans `instr`.

Indice 3

Utilise l'analyseur de l'exemple tel quel.

À toi, avec aide · réponse

Réponse

```
# (Analyseur de l'exemple)
erreurs, _ = Analyseur().analyser(parser("if true { let x = 1; }\nprint x;"))
print(erreurs)          # [(2, 7, "variable inconnue 'x'")]
erreurs, _ = Analyseur().analyser(parser("let x = 1;\nif true { print x; }"))
print(erreurs)          # []
```

La portée du bloc `if` est ouverte puis fermée : le `x` déclaré dedans disparaît à la sortie. C'est la **portée lexicale** : un nom n'existe que dans le bloc où il est déclaré (et ses blocs internes).

À toi, en solo

Contrôle des fonctions. Étends l'analyseur pour les fonctions :

1. `fun nom(params) { … }` : déclare `nom` (type `fonction`, avec son nombre de paramètres) dans la portée courante ; les paramètres sont déclarés dans la portée de la fonction ;
2. un appel `f(a, b)` doit viser une fonction connue avec le bon nombre d'arguments, sinon erreur positionnée ;
3. `return` hors d'une fonction est une erreur ;
4. un avertissement si une fonction n'a pas de `return` sur son chemin principal (simplification : aucun `Retour` au premier niveau du corps).

Teste avec 6 programmes.

À toi, en solo · réponse

Pistes de correction

Idée de réalisation (code abrégé) :

```
# Dans Analyseur :
#   self.fonctions = []         # pile de fonctions en cours (pour 'return')
# Info reçoit un champ 'arite' (None si ce n'est pas une fonction)
#
# instr(Fun) :
#   declarer(n.nom, "fonction", arite=len(n.params))        (erreur si déjà déclar
   ée)
#   ouvrir(); declarer chaque paramètre (type "?") ; self.fonctions.append(n)
#   analyser le corps ; self.fonctions.pop() ; fermer()
#   si aucun Retour au premier niveau : avertissement "fonction sans return"
# instr(Retour) : si not self.fonctions : erreur "return en dehors d'une fonction"
# type_de(Appel) :
#   si la cible est une Var inconnue -> erreur "fonction inconnue"
#   si info.type != "fonction"     -> erreur "ce n'est pas une fonction"
#   si len(args) != info.arite     -> erreur "attend N arguments, reçu M"
#   analyser aussi chaque argument (variables inconnues, etc.) ; type de retour "?
```

À toi, en solo · réponse

Pistes de correction (suite)

```
    "
```

Les tests couvrent : appel correct, mauvais nombre d'arguments, appel d'une variable non fonction, `return` global, fonction inconnue, récursion (la fonction doit être déclarée **avant** d'analyser son corps pour pouvoir s'appeler elle-même).

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
S'arrêter à la première erreur	L'utilisateur corrige une erreur à la fois	Collecter plusieurs erreurs
Erreurs en cascade	Une faute en produit dix	Type ? pour les sous-expressions invalides
Oublier de dépiler la portée	Fuite de variables entre blocs	Ouvrir/fermer à chaque bloc
Mélanger erreurs et avertissements	On bloque pour rien ou on laisse passer le grave	Deux niveaux de gravité

Mini-boss

Mini-boss

Le contrôleur de qualité de HashScript. Complète l'analyseur pour qu'il détecte, avec positions :

1. variables inconnues, redéclarations, types incompatibles, conditions non booléennes ;
2. appels de fonctions (existence, nombre d'arguments) et `return` hors fonction ;
3. **code mort** : instructions après un `return` dans le même bloc ;
4. **variables jamais utilisées** et **paramètres inutilisés** (avertissements) ;
5. **division par zéro constante** (`x / 0`) détectée statiquement.

Écris une suite d'au moins 20 programmes de test (valides et invalides) avec les messages attendus, puis exécute l'analyseur sur les programmes de la mission 10 du module pour montrer qu'ils n'ont aucune erreur.

Mini-boss · réponse

Pistes de correction

Un bon travail : une classe `Analyseur` avec `erreurs` et `avertissements` triés par position; la détection du code mort parcourt chaque bloc et signale la première instruction suivant un `Retour`; la division par zéro constante regarde si le diviseur (après pliage de constantes) est `Num(0)`; les tests utilisent un tableau `(source, erreurs_attendues, avertissements_attendus)`. L'exécution sur les programmes du capstone (factorielle, Fibonacci, fermetures) ne doit produire aucune erreur; si elle en produit, c'est généralement une portée de fonction mal gérée (la fonction doit voir ses propres paramètres et la fonction englobante).

Quiz éclair

Quiz éclair

1. L'analyse sémantique vérifie :

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

2. Une table des symboles associe :

A. chaque nom à ses informations (type, portée, position)
B. chaque jeton à sa position
C. chaque ligne à son numéro

3. Le typage statique vérifie les types :

A. pendant l'exécution seulement

Quiz éclair

Quiz éclair (suite)

B. avant l'exécution
C. jamais

4. 🔁 *Révision mission 04* : L'AST garde :

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

5. 🔁 *Révision mission 04* : Une optimisation doit toujours :

A. préserver le sens du programme
B. réduire le nombre de lignes
C. utiliser plus de mémoire

Quiz éclair · réponse

Réponses

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

Je réfléchis

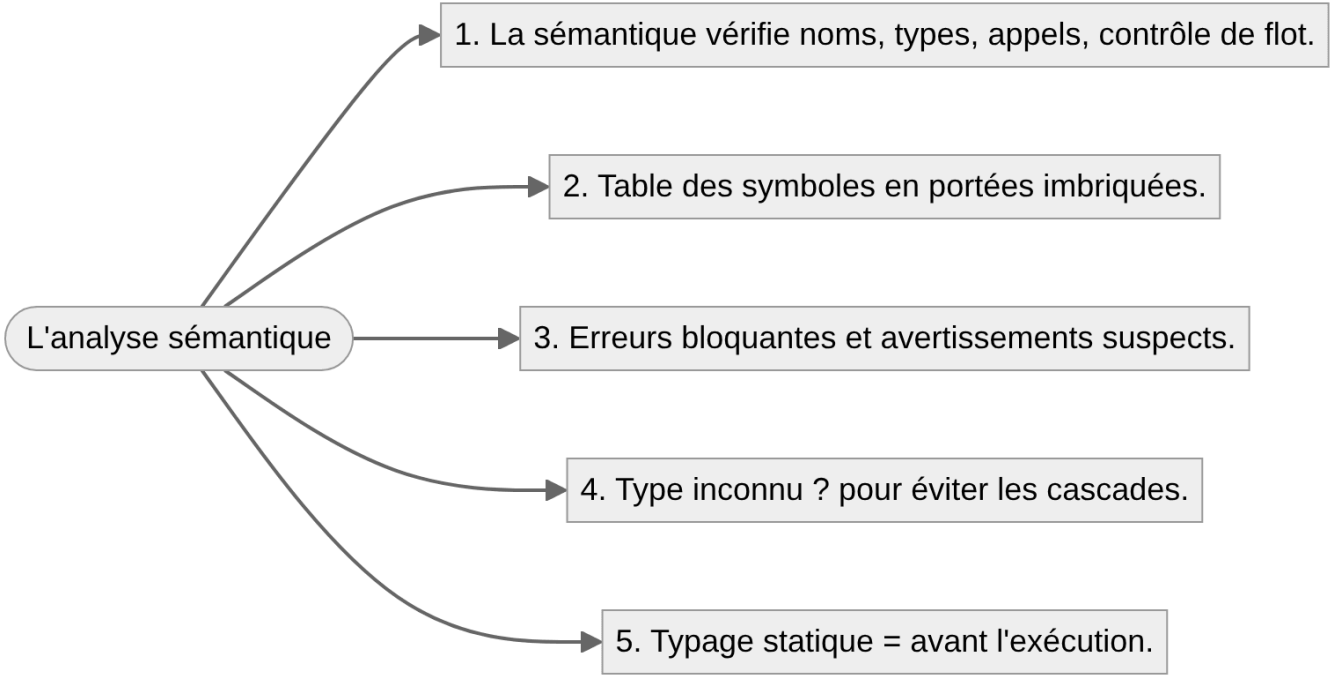
1. Quelle règle « de bon sens » (et non de forme) as-tu dû vérifier récemment dans ton travail ?
2. Pourquoi signaler plusieurs erreurs d'un coup est-il plus agréable pour l'utilisateur ?

À retenir

1. La sémantique vérifie noms, types, appels, contrôle de flot.
2. Table des symboles en portées imbriquées.
3. Erreurs (bloquantes) et avertissements (suspects).
4. Type inconnu ? pour éviter les cascades.
5. Typage statique = avant l'exécution.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Inspecteur de sens

Prochaine mission : L'interpréteur : faire vivre l'arbre

