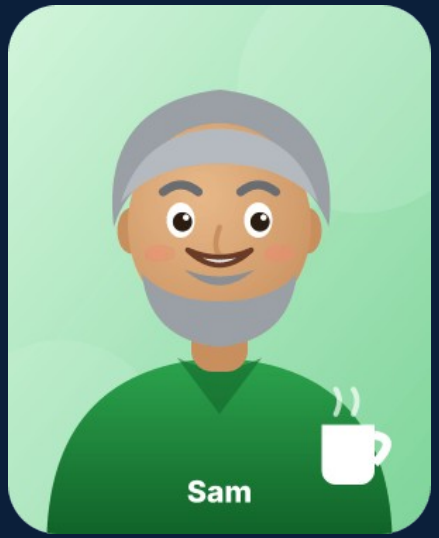
HEF-090 · MISSION 07 / 10

Environnement, portée et fermetures

Où vit une variable ? Qui peut la voir ? La réponse définit le comportement des fonctions.





Badge : Cartographe des noms

95 min · intermédiaire

L'histoire

HashScript sait faire des boucles mais pas encore des fonctions. Kodi ajoute `fun`. Un bug étrange apparaît : une fonction lit `x` et obtient la valeur de celui qui l'appelle, au lieu de celui qui l'a écrite.

> *« Tu as implémenté la portée dynamique sans le savoir, dit Sam. La plupart des langages modernes utilisent la portée lexicale. La différence est subtile, mais elle change tout, y compris les fermetures. »*



Tu implémentes les fonctions, la portée lexicale et les fermetures de HashScript.

Ta mission

À la fin, je serai capable de…

1. distinguer portée lexicale et portée dynamique avec un exemple
2. implémenter des environnements chaînés (parent)
3. implémenter appels de fonctions, `return` et récursion
4. expliquer une fermeture (closure) et la coder
5. gérer le masquage de variables (shadowing)

Défi d'échauffement

Défi d'échauffement

Tu lis dans un livre de cuisine : « Pour la sauce, utilise *le sel* ». Dans **ta** cuisine, il y a du sel fin ; dans la cuisine de ton voisin, du gros sel. Si tu envoies la recette au voisin, il utilise **son** sel. Maintenant, si tu écris sur la recette « le sel *de la maison qui a écrit la recette* », quel sel utilise-t-il ? Quelle est la différence entre « sel de celui qui exécute » et « sel de celui qui a écrit » ?

Le concept

L'environnement

Un **environnement** associe des noms à des valeurs. Pour gérer les portées imbriquées, chaque environnement a un **parent** :

```
env global        {x: 1, f: <fonction>}
   └─ env de f    {a: 10}            parent = env global
        └─ bloc   {t: 3}             parent = env de f
```

- **Définir** : dans l'environnement courant.
- **Chercher** : dans l'environnement courant, puis son parent, etc. jusqu'au global.
- **Affecter** : à l'environnement le plus proche qui contient déjà le nom.

Le concept

Portée lexicale ou dynamique

Quand une fonction utilise une variable qui n'est ni un paramètre ni une variable locale, **où** la chercher ?

Portée	Règle	Langages
Lexicale (statique)	dans les environnements où la fonction a été **écrite**	Python, JavaScript, Java, presque tous
Dynamique	dans l'environnement de **l'appelant**	anciens Lisp, shell, Perl (`local`)

```
let x = 1;
fun f() { return x; }
fun g() { let x = 2; return f(); }
print g();
```

Le concept

Portée lexicale ou dynamique (suite)

- Lexicale : `f` voit le `x` du programme où elle a été écrite → **1**.
- Dynamique : `f` voit le `x` de l'appelant `g` → **2**.

La portée lexicale rend un programme **lisible localement** : on sait ce que fait `f` en regardant son texte, sans connaître ses appelants.

La fermeture (closure)

Une **fermeture** est une fonction **accompagnée de l'environnement dans lequel elle a été créée**. Quand on l'appelle, ses variables libres se résolvent dans cet environnement, même si la fonction qui l'a créée est terminée.

Le concept

La fermeture (closure) (suite)

```
fun compteur() {
    let c = 0;
    fun incrementer() { c = c + 1; return c; }
    return incrementer;
}
let a = compteur();
a(); a();
print a();      # 3 : 'c' a survécu à la fin de compteur()
```

Les fermetures permettent : des compteurs et générateurs, des fonctions de configuration, des rappels, de l'encapsulation de données privées.

Implémentation

Le concept

Implémentation (suite)

```
class Environnement:
    def __init__(self, parent=None):
        self.vars, self.parent = {}, parent

    def trouver(self, nom):
        env = self
        while env is not None:
            if nom in env.vars:
                return env
            env = env.parent
        return None

class Fermeture:
    def __init__(self, fun, env):
        self.fun, self.env = fun, env       # la fonction ET son environnement de
   définition
```

Appeler une fonction :

Le concept

Implémentation (suite)

1. évaluer les arguments dans l'environnement de l'**appelant** ;
2. créer un nouvel environnement **dont le parent est celui de la fermeture** (lexical) ;
3. y définir les paramètres ;
4. exécuter le corps ; un `return` interrompt l'exécution (on utilise une exception interne `Retourne`).

Pour la portée **dynamique**, il suffit de choisir comme parent l'environnement de l'appelant.

La récursion

Une fonction se retrouve elle-même car son nom est défini dans l'environnement où elle a été déclarée, et son corps s'exécute dans un environnement dont ce parent contient son propre nom.

Le concept

Le masquage (shadowing)

Une variable déclarée dans une portée interne **masque** celle de même nom dans une portée externe, sans la modifier :

```
let x = 1;
if true { let x = 2; print x; }    # 2
print x;                            # 1
```

Les pièges

- **Capturer une variable de boucle** : toutes les fermetures créées dans une boucle partagent la même variable.
- **Récursion infinie** : il faut une limite de profondeur.

Le concept

Les pièges (suite)

- **Affecter à une variable non déclarée** : erreur, ou création implicite ? C'est un choix de conception (HashScript : erreur).

Exemple pas à pas

Problème : implémenter fonctions, fermetures et comparer portée lexicale / dynamique.

```
import operator
from hashscript import parser, Let, Assign, Print, Si, TantQue, Fun, Retour, Appel
   , Num, Bool, Var, Bin, 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 Env:
    def __init__(self, parent=None):
        self.vars, self.parent = {}, parent
    def trouver(self, nom):
        e = self
        while e is not None:
            if nom in e.vars: return e
```

Exemple pas à pas (suite)

```
            e = e.parent

class Retourne(Exception):
    def __init__(self, valeur): self.valeur = valeur

class Fermeture:
    def __init__(self, fun, env): self.fun, self.env = fun, env

class Interprete:
    def __init__(self, portee="lexicale", profondeur_max=200):
        self.portee, self.sortie, self.global_env = portee, [], Env()
        self.profondeur, self.profondeur_max = 0, profondeur_max

    def lancer(self, source):
        self.bloc(parser(source), self.global_env)
        return self.sortie

    def bloc(self, instrs, env):
        for i in instrs:
```

Exemple pas à pas (suite)

```
            self.executer(i, env)

    def executer(self, n, env):
        if isinstance(n, Let): env.vars[n.nom] = self.evaluer(n.expr, env)
        elif isinstance(n, Assign):
            cible = env.trouver(n.nom)
            if cible is None: raise ErreurLangage(f"affectation à une variable non
    déclarée {n.nom!r}", n.ligne, n.col)
            cible.vars[n.nom] = self.evaluer(n.expr, env)
        elif isinstance(n, Print): self.sortie.append(self.formater(self.evaluer(n
    .expr, env)))
        elif isinstance(n, Si): self.bloc(n.alors if self.evaluer(n.cond, env) els
    e n.sinon, Env(env))
        elif isinstance(n, TantQue):
            while self.evaluer(n.cond, env):
                self.bloc(n.corps, Env(env))
        elif isinstance(n, Fun): env.vars[n.nom] = Fermeture(n, env)       # on ca
    pture l'environnement
        elif isinstance(n, Retour): raise Retourne(self.evaluer(n.expr, env))
```

Exemple pas à pas

Exemple pas à pas (suite)

```
        else: self.evaluer(n, env)

    def evaluer(self, n, env):
        if isinstance(n, (Num, Bool)): return n.valeur
        if isinstance(n, Var):
            e = env.trouver(n.nom)
            if e is None: raise ErreurLangage(f"variable inconnue {n.nom!r}", n.li
    gne, n.col)
            return e.vars[n.nom]
        if isinstance(n, Neg): return -self.evaluer(n.expr, env)
        if isinstance(n, Bin):
            a, b = self.evaluer(n.gauche, env), self.evaluer(n.droite, env)
            if n.op == "/" and b == 0: raise ErreurLangage("division par zéro", n.
    ligne, n.col)
            return OPS[n.op](a, b)
        if isinstance(n, Appel):
            f = self.evaluer(n.fonction, env)
            if not isinstance(f, Fermeture): raise ErreurLangage("ce n'est pas une
    fonction", n.ligne, n.col)
```

Exemple pas à pas

Exemple pas à pas (suite)

```
            if len(n.args) != len(f.fun.params):
                raise ErreurLangage(f"{f.fun.nom} attend {len(f.fun.params)} argum
    ent(s), reçu {len(n.args)}", n.ligne, n.col)
            valeurs = [self.evaluer(a, env) for a in n.args]               # éval
    ués chez l'appelant
            parent = f.env if self.portee == "lexicale" else env            # LA
    différence
            local = Env(parent)
            for nom, v in zip(f.fun.params, valeurs): local.vars[nom] = v
            self.profondeur += 1
            if self.profondeur > self.profondeur_max:
                raise ErreurLangage("récursion trop profonde", n.ligne, n.col)
            try:
                self.bloc(f.fun.corps, local)
            except Retourne as r:
                return r.valeur
            finally:
                self.profondeur -= 1
            return None
```

Exemple pas à pas (suite)

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

    @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)

# --- récursion et fermetures ---
assert Interprete().lancer("fun f(n) { if n <= 1 { return 1; } return n * f(n - 1)
    ; } print f(6);") == ["720"]
compteur = '''
fun compteur() { let c = 0; fun inc() { c = c + 1; return c; } return inc; }
let a = compteur(); let b = compteur();
a(); a();
print a(); print b();
'''
```

Exemple pas à pas (suite)

```
print(Interprete().lancer(compteur))              # ['3', '1'] : a et b ont chacun
    leur propre 'c'

# --- portée lexicale vs dynamique ---
programme = "let x = 1; fun f() { return x; } fun g() { let x = 2; return f(); } p
   rint g();"
print("lexicale  :", Interprete("lexicale").lancer(programme))     # ['1']
print("dynamique :", Interprete("dynamique").lancer(programme))    # ['2']
```

Lecture :

- `a` et `b` sont deux fermetures issues du même `compteur`, mais avec **deux environnements distincts** (chaque appel de `compteur()` crée son propre `c`) ;
- en portée **lexicale**, `f` renvoie 1 (le `x` visible là où `f` est écrite) ; en **dynamique**, 2 (le `x` de l'appelant) ;
- une récursion de profondeur maximale est interceptée par une erreur claire.

À toi, avec aide

Écris en HashScript (puis exécute avec l'interpréteur de l'exemple) :

1. une fonction `additionneur(n)` qui renvoie une fonction ajoutant `n` à son argument ; vérifie que `additionneur(5)(10)` vaut 15 (l'appel chaîné est supporté par le parseur) ;
2. une fonction `fois_n(n)` qui renvoie une fonction multipliant par `n`, puis une fonction `composer(f, g)` qui renvoie une fonction appliquant `g` puis `f`.

Teste `composer(additionneur(1), fois_n(3))(4)` (attendu : 13).

À toi, avec aide · réponse

Indices progressifs

Indice 1

Une fonction peut déclarer une autre fonction à l'intérieur et la renvoyer avec `return nom;`.

Indice 2

Les paramètres sont des variables locales : les fonctions internes les capturent.

Indice 3

Pour composer : `fun composer(f, g) { fun h(x) { return f(g(x)); } return h; }`.

À toi, avec aide · réponse

Réponse

```
# (Interprete de l'exemple)
programme = '''
fun additionneur(n) { fun ajoute(x) { return x + n; } return ajoute; }
fun fois_n(n) { fun multiplie(x) { return x * n; } return multiplie; }
fun composer(f, g) { fun h(x) { return f(g(x)); } return h; }
print additionneur(5)(10);
let g = composer(additionneur(1), fois_n(3));
print g(4);
'''
assert Interprete().lancer(programme) == ["15", "13"]
print("fermetures composées ok")
```

Chaque appel à `additionneur`, `fois_n` et `composer` crée un nouvel environnement qui capture ses paramètres : `n`, `f`, `g` survivent aussi longtemps que les fonctions renvoyées.

À toi, en solo

Pile d'appels et traceback. Améliore l'interpréteur pour qu'une erreur d'exécution dans une fonction indique la **chaîne d'appels** :

```
erreur ligne 2, colonne 24 : division par zéro
  appelée depuis g (ligne 3)
  appelée depuis f (ligne 5)
  appelée depuis <programme> (ligne 7)
```

1. Maintiens une pile `(nom_de_fonction, ligne_de_l_appel)`.
2. Quand une `ErreurLangage` traverse un appel, ajoute-lui un attribut `pile` (copie de la pile à ce moment).
3. Écris `afficher_traceback(erreur)` et teste avec trois fonctions imbriquées.

À toi, en solo · réponse

Pistes de correction

```
# (Interprete de l'exemple, étendu)
class InterpreteTraceback(Interprete):
    def __init__(self, **kw):
        super().__init__(**kw)
        self.pile = []                                          # (nom, ligne de l'a
   ppel)

    def evaluer(self, n, env):
        if isinstance(n, Appel):
            f = None
            try:
                f = self.evaluer(n.fonction, env)
            except ErreurLangage:
                raise
            nom = f.fun.nom if isinstance(f, Fermeture) else "?"
            self.pile.append((nom, n.ligne))
            try:
                return super().evaluer(n, env)
            except ErreurLangage as e:
```

Pistes de correction (suite)

```
                if not hasattr(e, "pile"):
                    e.pile = list(self.pile)
                raise
            finally:
                self.pile.pop()
        return super().evaluer(n, env)

def afficher_traceback(e):
    lignes = [f"erreur ligne {e.ligne}, colonne {e.colonne} : {e.message}"]
    pile = getattr(e, "pile", [])
    for i in range(len(pile) - 1, -1, -1):
        appelant = pile[i - 1][0] if i > 0 else "<programme>"
        lignes.append(f"  appelée depuis {appelant} (ligne {pile[i][1]})")
    return "\n".join(lignes)

source = '''fun h(x) { return 10 / x; }
fun g(x) { return h(x - 1); }
fun f(x) { return g(x + 0); }
print f(1);'''
```

À toi, en solo · réponse

Pistes de correction (suite)

```
try:
    InterpreteTraceback().lancer(source)
except ErreurLangage as e:
    print(afficher_traceback(e))
```

La pile se construit à l'appel et se défait à la sortie (`finally`). L'erreur capture une **copie** de la pile au moment où elle se produit : c'est l'équivalent du *traceback* de Python.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Parent = appelant au lieu de définition	Portée dynamique involontaire, comportements surprenants	Une fermeture garde l'environnement où elle est définie
Un environnement partagé entre les appels	Les appels s'écrasent mutuellement (récursion fausse)	Un nouvel environnement par appel
Capturer la variable de boucle	Toutes les fermetures voient la dernière valeur	Créer une portée par itération
Pas de limite de profondeur	Plantage de la machine hôte (pile Python)	Compteur de profondeur et erreur claire

Mini-boss

Mini-boss

Récursion, mémoïsation et générateurs en HashScript. Écris, en HashScript, des programmes qui utilisent les fermetures :

1. une fonction `memoiser` n'est pas possible sans structure de données : à la place, écris `fibonacci` itératif avec une fermeture « état » (`a`, `b`) qui renvoie le terme suivant à chaque appel ;
2. un **générateur de nombres pairs** (fermeture avec compteur) ;
3. une **banque** : `compte(solde)` renvoie une fonction `operation(montant)` qui modifie le solde privé et le renvoie (les montants négatifs sont des retraits) ; deux comptes sont indépendants ;
4. une fonction d'**ordre supérieur** `appliquer_n(f, n, x)` qui applique `f` `n` fois ;
5. Vérifie chaque programme avec l'interpréteur lexical et montre au moins un cas où le mode dynamique donnerait un **résultat différent** (ou une erreur).

Mini-boss · réponse

Pistes de correction

Exemples de réponses (HashScript) :

```
fun suite() { let a = 0; let b = 1; fun suivant() { let r = a; let t = a + b; a =
   b; b = t; return r; } return suivant; }
let s = suite(); s(); s(); s(); print s();                    # 3

fun pairs() { let n = 0 - 2; fun suivant() { n = n + 2; return n; } return suivant
   ; }

fun compte(solde) { fun operation(m) { solde = solde + m; return solde; } return o
   peration; }
let c1 = compte(100); let c2 = compte(50); c1(20); c2(0 - 10); print c1(0); print
   c2(0);   # 120 40

fun appliquer_n(f, n, x) { let i = 0; while i < n { x = f(x); i = i + 1; } return
   x; }
fun double(x) { return x * 2; }
print appliquer_n(double, 5, 1);                             # 32
```

Mini-boss · réponse

Pistes de correction (suite)

En portée dynamique, `operation` chercherait `solde` dans l'environnement de l'appelant (le programme principal), où `solde` n'existe pas : erreur « variable inconnue » — ce qui montre que **les fermetures dépendent de la portée lexicale**.

Quiz éclair

Quiz éclair

1. En portée lexicale, une variable libre d'une fonction est cherchée dans :

A. l'environnement où la fonction a été définie
B. l'environnement de l'appelant
C. la mémoire globale uniquement

2. Une fermeture est :

A. une fonction sans paramètre
B. une fonction accompagnée de son environnement de définition
C. un type de boucle

3. Le masquage (shadowing) signifie :

A. une variable est supprimée

Quiz éclair

Quiz éclair (suite)

B. une variable devient privée
C. une variable interne cache une variable externe de même nom, sans la modifier

4. 🔁 *Révision mission 06* : Un interpréteur d'arbre exécute un programme en :

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

5. 🔁 *Révision mission 06* : Une limite de pas protège contre :

A. les erreurs de syntaxe
B. les boucles infinies
C. les variables inutilisées

Quiz éclair · réponse

Réponses

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

Je réfléchis

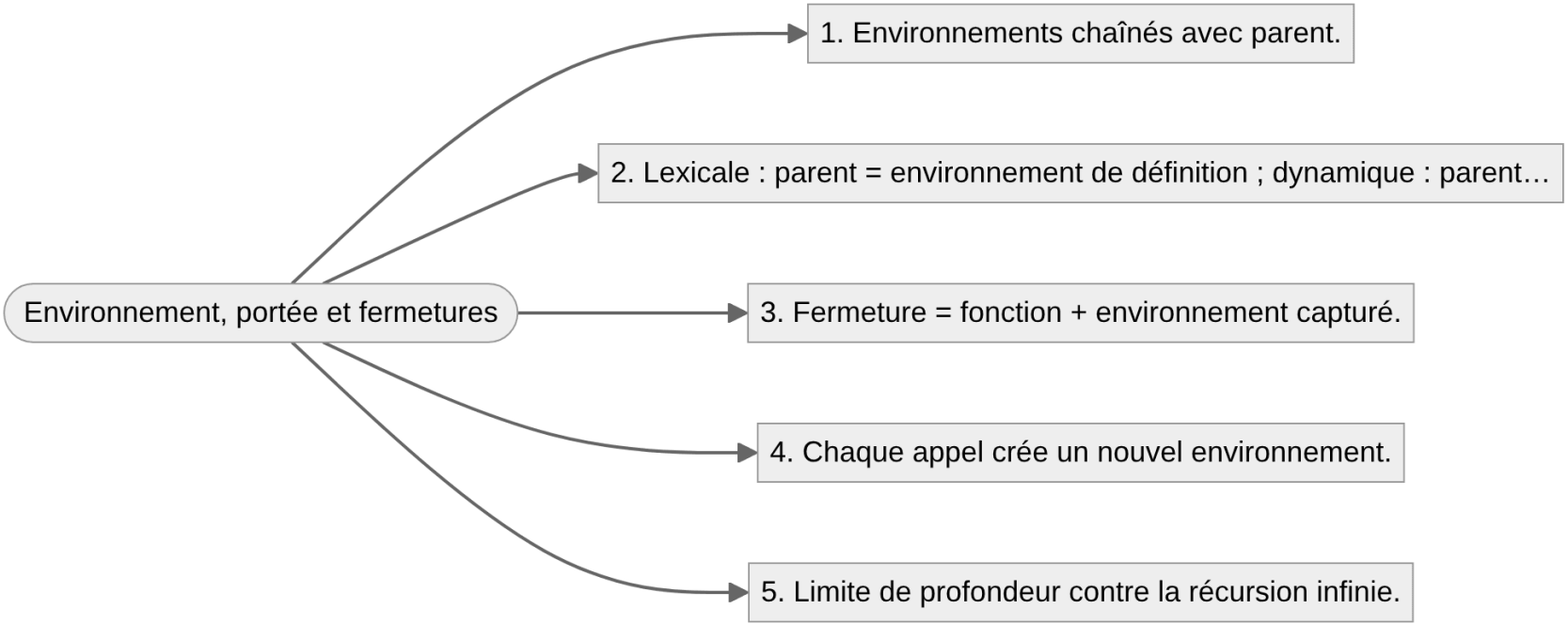
1. Dans ta vie, où une information dépend-elle du contexte où elle a été écrite plutôt que du contexte où elle est lue ?
2. Quels usages des fermetures rencontres-tu déjà dans ton travail (rappels, configuration, décorateurs) ?

À retenir

1. Environnements chaînés avec parent.
2. Lexicale : parent = environnement de définition ; dynamique : parent = appelant.
3. Fermeture = fonction + environnement capturé.
4. Chaque appel crée un nouvel environnement.
5. Limite de profondeur contre la récursion infinie.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Cartographe des noms

Prochaine mission : Bytecode et machine virtuelle

