PLP

Semantic Analysis

02 December 2025

Table des matières

1 Semantic Analysis

La sémantique d'un langage de programmation concerne la signification des constructions syntaxiques. L'analyse sémantique vérifie que les programmes respectent les règles sémantiques du langage, au-delà de la simple syntaxe.

Cette phase effectue des vérifications comme :

- **Name analysis** : Vérifie que toutes les variables et fonctions utilisées sont déclarées
- **Type checking** : Assure la compatibilité des types dans les opérations

1.1 Semantic Checks

Les validations sémantiques courantes incluent :

- Variables lues avant initialisation
- Labels dupliqués dans un switch
- Réaffectation de constantes
- Exhaustivité du pattern matching
- Visibilité des méthodes invoquées

1.2 Semantic Rules

Les règles sémantiques se divisent en deux catégories :

- **Static rules** : Vérifiées à la compilation (types, portée des variables)
- **Dynamic rules** : Vérifiées à l'exécution (division par zéro, accès ressources)

1.3 Warnings vs Errors

Semantic errors : Violations des règles sémantiques qui empêchent la compilation

- Variables non déclarées, types incompatibles, violations de portée

Warnings : Problèmes potentiels qui ne bloquent pas la compilation

- Variables non utilisées, code inaccessible, conversions de types implicites

2 Name Analysis

L'analyse de noms vérifie que toutes les références aux identificateurs sont valides.

2.1 Scope

La **portée** (scope) d'un identificateur définit la région du code où il est accessible. Les langages utilisent différentes stratégies de portée :

- **Portée lexicale** (lexical/static scope) : La portée est déterminée par la structure du code source
- **Portée dynamique** (dynamic scope) : La portée dépend de l'ordre d'exécution

2.2 Nested Scopes

Les portées peuvent être imbriquées. Une portée interne peut accéder aux variables de la portée externe, mais pas l'inverse.

```
int x = 10;
{
    int y = 20;
    // x et y accessibles ici
}
// seul x est accessible ici
```

2.3 Shadowing

Le **shadowing** se produit quand une variable dans une portée interne a le même nom qu'une variable dans une portée externe. La variable interne « masque » la variable externe.

```
int x = 10;
{
    int x = 20; // shadowing
    System.out.println(x); // affiche 20
}
System.out.println(x); // affiche 10
```

2.4 Duplicated Definitions

Les définitions dupliquées dans la même portée sont généralement interdites :

```
int x = 10;
int x = 20; // erreur : x déjà défini
```

2.5 Symbol Table

La **table des symboles** (symbol table) est une structure de données utilisée pour stocker les informations sur les identificateurs :

- Nom de la variable
- Type
- Portée
- Adresse mémoire (pour la génération de code)

Structure typique : dictionnaire avec recherche dans les portées imbriquées.

3 Type Systems

Un système de types définit les types de données disponibles dans un langage et les règles régissant leur utilisation.

3.1 Types

Les types peuvent être :

- **Primitifs** : int, float, bool, char
- **Composés** : arrays, structs, classes
- **Fonction** : type des paramètres et du retour

3.2 Static vs Dynamic Typing

Static typing : Les types sont vérifiés à la compilation

- Avantages : Détection précoce d'erreurs, optimisation
- Exemples : Java, C++, Rust

Dynamic typing : Les types sont vérifiés à l'exécution

- Avantages : Flexibilité, moins de code
- Exemples : Python, JavaScript, Ruby

3.3 Strong vs Weak Typing

Strong typing : Conversions de types explicites requises

- Exemple : Python (pas de conversion implicite int → str)

Weak typing : Conversions implicites fréquentes

- Exemple : JavaScript (« 5 » + 3 donne « 53 »)

3.4 Type Inference

L'**inférence de types** permet au compilateur de déduire automatiquement les types sans annotations explicites.

```
val x = 42        // inféré comme Int
val y = "hello"   // inféré comme String
```

4 Type Checking

Le **type checking** vérifie que les opérations sont effectuées sur des types compatibles.

4.1 Typing Rules

Les règles de typage définissent comment les types sont assignés aux expressions. Notation formelle :

$$\Gamma \vdash e : \tau$$

Signifie : « Dans l'environnement Γ, l'expression e a le type τ »

4.1.1 Basic Rules

Literals :

$$\Gamma \vdash n : \text{int} \qquad \Gamma \vdash \text{true} : \text{bool}$$

Variables :

$$(x : \tau) \in \Gamma \Longrightarrow \Gamma \vdash x : \tau$$

Addition :

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}$$

4.1.2 Function Rules

Application :

$$\frac{\Gamma \vdash f : \tau_1 \to \tau_2 \quad \Gamma \vdash e : \tau_1}{\Gamma \vdash f(e) : \tau_2}$$

Abstraction :

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

4.2 Derivation Trees

Les arbres de dérivation montrent l'application des règles de typage pour vérifier qu'une expression est bien typée.

```
Γ ⊢ e₁ : int    Γ ⊢ e₂ : int
──────────────────────────────
     Γ ⊢ e₁ + e₂ : int
```

4.3 Type Errors

Erreurs de typage courantes :

- Type mismatch : `int x = "hello";`
- Fonction appelée avec mauvais arguments
- Opération non supportée : `"hello" - 5`
- Retour de type incorrect

5 Type Inference

L'inférence de types permet de déduire automatiquement les types sans annotations.

5.1 Hindley-Milner Type System

Système d'inférence de types utilisé dans ML, Haskell, OCaml :

- Inférence complète sans annotations
- Polymorphisme paramétrique
- Types principaux (most general type)

5.2 Algorithm W

Algorithme d'inférence de types en deux phases :

1. **Génération de contraintes** : Parcourt l'AST et crée des équations de types
2. **Unification** : Résout les contraintes pour trouver les types

5.3 Type Variables

Variables de type (α, β) représentent des types inconnus :

```
id :: α → α
id x = x
```

5.4 Type Constraints

Les contraintes de types sont des équations à résoudre :

- `α = int`
- `α = β → γ`
- `β = bool`

5.5 Unification

L'**unification** résout les contraintes de types en trouvant une substitution qui rend les types égaux.

Algorithme d'unification :

1. Si `α = α`, succès
2. Si `α = τ` où α n'apparaît pas dans τ, substituer α par τ
3. Si `τ₁ → τ₂ = τ₃ → τ₄`, unifier τ_1 avec τ_3 et τ_2 avec τ_4
4. Sinon, échec

Occurs check : Empêche les types récursifs infinis comme `α = α → β`

5.6 Exemple Complet

```
let id = λx. x in id 5
```

1. Génération contraintes :
   - `id : α → α`
   - `5 : int`
   - Application : `α = int`
2. Unification : `α = int`
3. Type final : `id : int → int`, résultat : `int`