PLP

Interpreters

28 January 2026

Table des matières

1 Interpreters

Les interpreters se basent sur les étapes d'analyse lexicale, syntaxique et sémantique pour exécuter du code source. Ils **traduisent** le code source en instructions exécutables sans passer par une phase de compilation préalable.

Le code source est lu ligne par ligne, ou déclaration par déclaration, et chaque instruction est immédiatement interprétée et exécutée. Cela permet une exécution plus rapide du code, mais peut entraîner des performances moindres par rapport aux programmes compilés.

> **i Info**
>
> Les interpreters sont souvent utilisés pour les langages de script et les langages de programmation dynamiques, tels que Python, JavaScript et Ruby.

1.1 AST-based Interpreters

Les interpreters basés sur l'AST (Abstract Syntax Tree) construisent un arbre de syntaxe abstraite à partir du code source, puis parcourent cet arbre pour exécuter les instructions.

En d'autres mots, il implémente l'analyse lexicale et syntaxique pour construire l'AST, puis utilise une fonction d'évaluation qui visite chaque nœud de l'arbre et exécute les opérations correspondantes.

- Un **lexer** permet à un interpréteur de diviser le code source en séquence d'unités lexicales (tokens) significatives.
- Un **parser** analyse la séquence de tokens pour construire l'AST, qui représente la structure hiérarchique du code source.
- Une **fonction d'évaluation** parcourt l'AST et exécute les opérations définies par chaque nœud.

1.2 REPL

Un REPL (Read-Eval-Print Loop) est un environnement interactif qui permet aux utilisateurs de saisir des expressions ou des commandes, de les évaluer immédiatement et d'afficher les résultats.

- **Read** : Lit l'entrée de l'utilisateur.
- **Eval** : Évalue l'expression ou la commande.
- **Print** : Affiche le résultat de l'évaluation.
- **Loop** : Répète le processus pour chaque nouvelle entrée.

```
ghci> 1 + 2
3
```

1.3 Pros and cons

Les langages interprétés offrent plusieurs avantages et inconvénients par rapport aux langages compilés. Ils sont souvent **plus faciles à apprendre**, **plus rapide à utiliser**, **indépendant de la plateforme** et **plus facile à déboguer**. Cependant, ils peuvent être **plus lents** en termes de performance, **moins optimisés** et **moins adaptés** aux applications nécessitant une haute performance.

2 Evaluating expressions

Une expression est un combinaison de valeurs, variables, opérateurs et fonctions qui sont évaluées pour produire une nouvelle valeur.

2.1 Representing expressions

De manière générale, une expression peut être représenté epar un AST (Abstract Syntax Tree). Chaque nœud de l'arbre représente une opération ou une valeur, et les branches représentent les relations entre ces opérations et valeurs.

Dans cet exemple nous avons l'expression $(2 + 3) * 4$

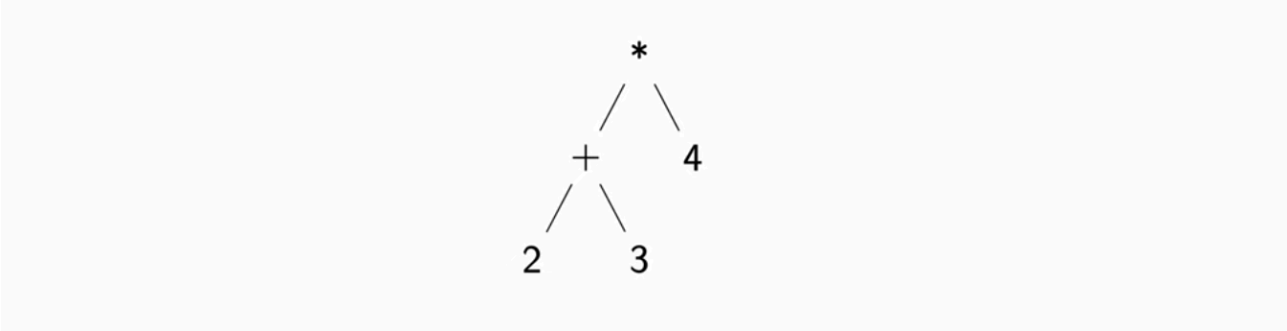

*Fig. 1. – Capture des slides du cours – Représentation AST de l'expression (2+3)*4*

```
data Expr = Const Int
          | Binary Expr Char Expr
```

2.2 Evaluating the AST

Pour évaluer les expression représentées par un AST, nous pouvons définir une fonction récursive qui parcourt l'arbre et calcule la valeur de chaque nœud en fonction de son type.

Pour le faire en langage fonctionnel (Haskell), nous pouvons définir une fonction `eval` qui prend un nœud de l'AST en entrée et retourne la valeur évaluée.

```haskell
eval :: Expr -> Int
eval (Const n) = n
eval (Binary left op right) =
  case op of
    '+' -> eval left + eval right
    '-' -> eval left - eval right
    '*' -> eval left * eval right
    _ -> error "Unknown operator"
```

On utilise alors la récursion pour évaluer les sous-expressions, puis on applique l'opérateur au résultat des sous-expressions.

2.3 Runtime errors

Lors de l'évaluation des expressions, des erreurs d'exécution (runtime errors) peuvent survenir. Ces erreurs peuvent être causées par diverses raisons, telles que la division par zéro, l'accès à des variables non définies ou des opérations invalides.

```haskell
eval :: Expr -> Int
eval (Binary left '/' right) =
  case (eval left, eval right) of
    (_, 0) -> error "Division by zero"
    (l, r) -> l `div` r
```

3 Statements and state

Un statement (instruction) est une unité de code qui effectue une action, telle que l'affectation d'une valeur à une variable, la déclaration d'une fonction ou le contrôle du flux d'exécution. Elle est la plus petite unité exécutable dans un programme.

3.1 Executing statements

Différemment des expressions, les statements ne produisent pas de valeur, mais modifient l'état du programme ou effectuent des actions.

3.2 Environment

Un environnement est une structure de données qui stocke les associations entre les variables et leurs valeurs dans un programme. Il permet de suivre l'état des variables au fur et à mesure de l'exécution du programme.

Un environnement est souvent représenté comme une pile ou un dictionnaire, où chaque entrée associe un nom de variable à sa valeur actuelle.

```
type Env = Map String Value
```

La fonction d'évaluation doit pouvoir accéder et modifier l'environnement lors de l'exécution des statements.

```
eval :: Expr -> Env -> Value
```

3.3 Representing state

L'état d'un programme fait référence aux valeurs actuelles des variables et structure de données se trouvant dans la mémoire à un moment donné de l'exécution.

1. L'état change continuellement au fur et à mesure de l'éxécution des instructions.
2. À tout moment, l'état du programme représente une « photo » des valeurs des variables et des structures de données.

```
data State = State {
  globals :: Env,
  locals :: Env
}
exec :: Stmt -> State -> State
```

4 Control flow

Le contrôle de flux (control flow) fait référence à l'ordre dans lequel les instructions d'un programme sont exécutées. Il permet de diriger l'exécution du programme en fonction de conditions, de boucles et d'autres structures de contrôle. Il est déterminé par les structures de contrôle telles que:

- Les instructions conditionnelles (if, else)
- Les boucles (for, while)
- Les appels de fonctions et les retours

4.1 Conditional execution

L'exécution conditionnelle permet d'exécuter différentes sections de code en fonction de conditions spécifiques. Cela est généralement réalisé à l'aide d'instructions conditionnelles telles que `if`, `else if` et `else`.

Nous pouvons aussi utiliser le `switch` lorsque nous avons besoin de comparer une variable à plusieurs valeurs possibles.

4.2 Loops

Les boucles permettent d'exécuter un bloc de code de manière répétée tant qu'une condition spécifiée est vraie. Les boucles sont utilisées pour itérer sur des collections de données, effectuer des calculs répétitifs ou exécuter des tâches jusqu'à ce qu'une certaine condition soit remplie.

4.3 Function calls

Les appels de fonctions permettent d'exécuter des blocs de code réutilisables définis dans des fonctions. Lorsqu'une fonction est appelée, le contrôle de l'exécution est transféré à la fonction, qui peut prendre des arguments, effectuer des opérations et retourner une valeur.

4.4 Short-circuit evaluation

L'évaluation à court-circuit (short-circuit evaluation) est une technique utilisée dans les expressions logiques pour optimiser l'évaluation des opérateurs logiques `AND` (&&) et `OR` (||). Avec cette technique, l'évaluation s'arrête dès que le résultat final peut être déterminé sans évaluer toutes les parties de l'expression.

On peut voir cela un peu comme une forme de `lazy evaluation`, où les parties de l'expression ne sont évaluées que lorsque cela est nécessaire pour déterminer le résultat final.

5 Functions and calls

Une fonction est un bloc de code autonome qui effectue une tâche spécifique. Elle peut prendre des entrées (paramètres), exécuter des opérations et retourner une sortie (valeur de retour).

Quand une fonction est appelée:

1. Les arguments sont évalués et passés aux paramètres de la fonction.
2. La définition de la fonction est récupérée dans l'environnement global ou local.
3. Un lien est créé entre les paramètres et l'environnement d'appel.

5.1 Function declaration

La déclaration d'une fonction définit son nom, ses paramètres et le corps de la fonction. Elle permet de créer une fonction réutilisable qui peut être appelée à différents endroits dans le programme.

```
data Decl = ...
          | Fun String [String] Expr
```

5.2 Closures

Une closure est une valeur « runtime » représentant une fonction et leur environnement, incluant les variables et bindings accessibles au moment de la création de la fonction.

- Lors de l'évaluation d'une fonction, il est essentiel pour l'interpréteur de capturer l'environnement dans lequel la fonction a été définie.
- Cela permet à la fonction d'accéder aux variables et bindings de son environnement même lorsqu'elle est appelée dans un contexte différent.

5.3 Returning values

Le retour de valeurs dans les fonctions permet de transmettre des résultats d'une fonction à l'appelant. Lorsqu'une fonction atteint une instruction de retour, l'exécution de la fonction s'arrête et la valeur spécifiée est renvoyée à l'appelant.

5.3.1 Native functions

Les fonctions natives sont des fonctions intégrées au langage de programmation ou à l'environnement d'exécution. Elles sont généralement optimisées pour des performances élevées et fournissent des fonctionnalités de base, telles que les opérations mathématiques, la manipulation de chaînes de caractères et les opérations sur les collections de données.