Interprétation

Dans le développement des programmes, il est souvent utile de travailler avec un interprète au lieu d'un compilateur.

L'interprétation se fait immédiatement, ce qui réduit le temps d'attente en lançant le compilateur (aspect psychologique important).

Le temps d'exécution est en général quand même plus long que celui du code compilé, comme ce dernier est normalement optimisé.

Par contre, l'interprétation est indépendante de la plate-forme.

Java adopte une démarche combinée: les programmes sont d'abord compilés (`javac`) et ensuite interprété (`java`) sur la JVM (java virtual machine). La JVM est une **machine abstraite**: elle n'existe qu'en forme de programmes installés sur des architectures concrètes.

Le code purement compilé est avantageux surtout quand un programme manipule des grandes quantités de données. Dans ce cas, on peut obtenir des effets visibles par une bonne adaptation à la machine concrète.

Survol de cette période:

- Sémantique opérationnelle
- Interprétation des expressions
- Interprétation des commandes
- Interprétation des processus parallèles

Un exemple d'interprétation

L'interprétation exécute les programmes directement.

Exemple: `if (x = 1) then a := 1 else a := 2`

- D'abord, on évalue la condition.
- Si le résultat est `true`, on interprète `a := 1`.
- Sinon, on interprète `a := 2`.

En pratique, on peut réaliser cette démarche avec des visiteurs.

Expressions: L'interprétation retourne comme résultat la valeur de l'**évaluation**.

Commandes: L'interprétation n'a pas de résultat.

Pour le projet, cela implique: deux méthodes de visite.

```
public class Interpreter implements Tree.Visitor {

    int result;

    static int visitExp(Tree tree) {
        Interpreter v = new Interpreter();
        tree.apply(v);
        return v.result;
    }

    static int visitStat(Tree tree) {
        Interpreter v = new Interpreter();
        tree.apply(v);
    }

    ...
}
```

Expressions

```
public void caseNum(Tree.Num tree) {
    result = num;
}

public void caseId(Tree.Num tree) {
    result = scope.lookup(tree.id);
}

public void caseOp(Tree.Op tree) {
    int l = visitExp(tree.left);
    int r = visitExp(tree.right);
    switch (tree.op) {
    case PLUS: result = l + r; break;
    ...
    }
}
```

Commandes

```
public void caseAssign(Tree.Assign tree) {
    int res = visitExp(tree.exp);
    scope.update(a, res);
}

public void caseCond(Tree.Cond tree) {
    int res = visitExp(tree.cond);
    if (res) visitStat(tree.then);
    else visitStat(tree.els);
}

...
```

Sémantique opérationnelle

Introduit par Gordon Plotkin en 1981. La sémantique opérationnelle décrit ce que fait un programme lors de son exécution sur une **machine abstraite**.

D'autres techniques de description: **sémantique dénotationnelle**, **sémantique axiomatique**. Voir par exemple: G. Winskel, *The Formal Semantics of Programming Languages*, MIT Press, 1993.

L'**environnement** σ représente les portées. Sur des **configurations** $\langle a, \sigma \rangle$ on définie des relations **d'évaluation** (expressions) et **d'exécution** (commandes).

Système de règles:

- Un nombre s'évalue sur lui-même: $\langle n, \sigma \rangle \longrightarrow n$
- Un identificateur s'évalue sur la valeur correspondante dans l'environnement:

$$\langle x, \sigma \rangle \longrightarrow \sigma(x)$$

- Une addition s'évalue sur la somme des résultats des deux composantes:

$$\frac{\langle e_1, \sigma \rangle \longrightarrow n_1 \quad \langle e_2, \sigma \rangle \longrightarrow n_2}{\langle e_1 +_o e_2, \sigma \rangle \longrightarrow n_1 + n_2} \quad \text{opérateur } +_o \text{ du langage}$$

- Une déclaration rajoute la variable à l'environnement; une assignation modifie l'entrée pour la variable spécifiée:

$$\frac{\langle e, \sigma \rangle \longrightarrow n \quad x \notin \sigma}{\langle x := e, \sigma \rangle \longrightarrow \sigma \cup [x, m]} \qquad \frac{\langle e, \sigma \rangle \longrightarrow n \quad x \in \sigma}{\langle x := e, \sigma \rangle \longrightarrow \sigma[m/x]}$$

- Une boucle est exécuté jusqu'à ce que la condition soit `false`:

$$\frac{\langle b, \sigma \rangle \longrightarrow \texttt{false}}{\langle \textbf{while } b\ s, \sigma \rangle \longrightarrow \sigma} \qquad \frac{\langle b, \sigma \rangle \longrightarrow \texttt{true} \quad \langle s, \sigma \rangle \longrightarrow \sigma' \quad \langle \textbf{while } b\ s, \sigma' \rangle \longrightarrow \sigma''}{\langle \textbf{while } b\ s, \sigma \rangle \longrightarrow \sigma''}$$

- ...

Dérivations

Une **dérivation** s'écrit sous forme de **tableau (arbre)**. Les branches d'un tel tableau sont marquées par des **axiomes** (règles sans prémisses) ou par des faits déjà prouvés par d'autres tableaux.

Exemple: Évaluation d'une expression.

$$\frac{\langle a, [(a,1),(b,2)]\rangle \longrightarrow 1 \qquad \langle 3, [(a,1),(b,2)]\rangle \longrightarrow 3}{\langle a+3, [(a,1),(b,2)]\rangle \longrightarrow 4}$$

Exemple: Exécution d'une commande.

$$\frac{\dfrac{\langle a, [(a,1),(b,2)]\rangle \longrightarrow 1 \qquad \langle 3, [(a,1),(b,2)]\rangle \longrightarrow 3}{\langle a+3, [(a,1),(b,2)]\rangle \longrightarrow 4}}{\langle b := a+3, [(a,1),(b,2)]\rangle \longrightarrow [(a,1),(b,4)]}$$

Equivalence

On dit que deux expressions/programmes sont **équivalents** si sur tous les environnements d'entrée contenant toutes les variables libres, ils produisent le même résultat:

$$\begin{array}{lcl} e \sim e' & := & \forall \sigma, n.\ var(e,e') \subseteq dom(\sigma) \Rightarrow (\langle e,\sigma\rangle \longrightarrow n \Leftrightarrow \langle e',\sigma\rangle \longrightarrow n) \\ s \sim s' & := & \forall \sigma, \sigma'.\ var(s,s') \subseteq dom(\sigma) \Rightarrow (\langle s,\sigma\rangle \longrightarrow \sigma' \Leftrightarrow \langle s',\sigma\rangle \longrightarrow \sigma') \end{array}$$

Exercices:

1. Montrez que les programmes `a := 1; a := a + 1` et `a := 2` sont équivalents.
2. Est-ce que les programmes seraient équivalents dans un contexte parallèle?
3. Montrez que les programmes `a := 1; a := a + 1` et `b := 0; a := 2` ne sont pas équivalents.

Processus

Il est évident dans les derniers exemples que la sémantique discutée plus haut n'est pas assez raffinée pour les procesus parallèles.

Un processus contenant la commande `a := 0` exécuté en parallèle pourrait interférer avec la première séquence de commandes.

Pour décrire des processus, on se base sur une **sémantique à petits pas**. En général, on garde les environnements dans le cas des expressions. D'une part, cela permet d'un système de règles plus compact (moins de règles pour les opérateurs). D'autre part, cela permet de décrire les **effets de bord** (modifications de l'environnement pendant l'évaluation des expressions).

Système de règles:

- Un nombre s'évalue sur lui-même: $\langle n, \sigma\rangle \longrightarrow_1 \langle n, \sigma\rangle$
- Un identificateur s'évalue sur la valeur correspondante dans l'environnement:

$$\langle x, \sigma\rangle \longrightarrow_1 \langle \sigma(x), \sigma\rangle$$

- Evaluation des expressions; en cas de deux valeurs retour du résultat:

$$\langle n +_o m, \sigma\rangle \longrightarrow_1 \langle n + m, \sigma\rangle$$

$$\frac{\langle e_1, \sigma\rangle \longrightarrow_1 \langle e_1', \sigma'\rangle}{\langle e_1 +_o e_2, \sigma\rangle \longrightarrow_1 \langle e_1' + e_2, \sigma'\rangle} \qquad \frac{\langle e_2, \sigma\rangle \longrightarrow_1 \langle e_2', \sigma'\rangle}{\langle e_1 +_o e_2, \sigma\rangle \longrightarrow_1 \langle e_1 + e_2', \sigma'\rangle}$$

- Une déclaration rajoute la variable à l'environnement; une assignation modifie l'entrée pour la variable spécifiée:

$$\frac{x \notin \sigma}{\langle x := m, \sigma\rangle \longrightarrow_1 \sigma' \cup [x, m]} \qquad \frac{x \in \sigma}{\langle x := m, \sigma\rangle \longrightarrow_1 \sigma[m/x]}$$

$$\frac{\langle e, \sigma\rangle \longrightarrow_1 \langle e', \sigma'\rangle}{\langle x := e, \sigma\rangle \longrightarrow_1 \langle x := e', \sigma'\rangle}$$

- ...

Exercices

[1] Complétez le système de règles de la sémantique opérationnelle utilisant $\longrightarrow$ pour les commandes du projet.

[2] Complétez le système de règles de la sémantique opérationnelle utilisant $\longrightarrow_1$ pour les processus du projet.

[3] Effectuez des dérivations dans les deux systèmes.

[4] Expliquez pourquoi les deux commandes de la page 56 sont équivalentes dans la sémantique à grands pas et non dans la sémantique à petits pas.

[5] Les pairs d'expressions ou commandes suivants sont-ils équivalents? Expliquez votre réponse.

- `a + 1 + 1` et `a + 2`;
- `a + b` et `b + a`;
- `if (x = 0) then a := 3 else a := 3` et `a := 3`;
- `i := 1` et `while (i = 0) { i := 1 }`.

[6] Réalisez un interprète d'expressions et commandes pour le projet. Voir pages web du cours.