Analyse sémantique

Les langages de programmation ne sont pas non-contextuels. Par exemple, une variable doit être déclarée ailleurs *dans le contexte* avant son utilisation.

En pratique, on spécifie un sur-ensemble du langage avec une grammaire non-contextuelle (voir analyse syntaxique), et on élimine les programmes illégaux plus tard pendant l'**analyse sémantique**.

La deuxième tâche d'un analyseur sémantique est la construction de la **table de symboles**.

Survol de cette période:

- Symboles
- Types
- Portées (tables de symboles)

Ce cours traite pas le sujet des **attributs**. En général, les arbres pourraient contenir un champ pour chacun des attributs, qui est rempli par l'analyseur sémantique. Des exemples typiques d'attributs sont le *type* d'une expression, le *symbole* produit par la déclaration d'une variable ou d'une fonction, ou la *table des symboles* produit par une suite de déclarations.

Symboles

Pour chaque ne déclaration (par exemple, `int x;` en Java) ou définition (par exemple, `int x = 3;`), le compilateur produit un **symbole**.

Un symbole est une structure de donnée qui contient toutes les informations sur un identificateur (variable, fonction, processus, ...) que le compilateur doit connaître.

Un symbole porte un nom. Ceci peut être le nom de l'identificateur (voir projet), mais aussi un nom créé par le compilateur. Dans le dernier cas, il est parfois utile de stocker aussi le nom original dans le symbole.

En général, un symbole a un type. La classe des types est généralement construite par analogie avec la classe d'arbres (classe abstraite `Type`, sous-classes concrètes).

Il peut être nécessaire de parcourir les symboles d'une portée dans l'ordre de leur déclaration. Dans ce cas, on les lie linéairement avec un champ `next`.

```
class Symbol {
   Symbol next;   // pour parcourir les symboles d'une portee
   String name;   // nom du symbole
   Type type;     // type du symbole
}
```

Dans le projet, nous n'avons pas besoin d'un champ pour le type, mais pour interpréter les programmes, nous rajoutons un champ portant la valeur d'une variable.

Types

Chaque déclaration ou définition produit un **type**. La définition

```
int x = 3;
```

par exemple, introduit une variable *de type* `int`. La définition

```
int add(int i, int j) { return i + j; }
```

introduit une fonction de type `(int, int) => int`: elle prend deux arguments de type `int` et retourne un résultat de type `int`.

Pour le type, les noms des paramètres sont redondants.

Une grammaire pour des types

$$\begin{array}{lcl} \text{Type} & \rightarrow & \text{IntType} \\ & | & \text{FunType} \;\; \text{Scope} \;\; \text{Type} \end{array}$$

par exemple, peut être réalisé comme suit:

```
abstract class Type {
    static class IntType extends Type {}
    static class FunType extends Type {
        Symbol params;   // champ next lie plusieurs parametres
        Type resType;
    }
}
```

Notez la correspondance avec la syntaxe abstraite (grammaires abstraites, arbres abstraits).

Portées (tables de symboles)

Une portée `scope` décrit les zones de visibilité des identificateurs. Les portées sont généralement imbriquées; un champ `outer` se réfère à la portée directement englobante.

```
int i = 3;                                          | i(1)    | f
int j = 1;                                          |    | j  |
int f(int k, int l) {                     | k   | l |    |    |
  i := l;                                 |     |   |    |    |
  for (int i = 1; i < k; i++) {           |     |        |    |    | i(2)
    j = i * j;                            |     |        |    |    |
  }                                       |     |        |    |    |
  return j + l;                           |     |   |    |    |
}                                                   |    |    |
f(i, j);                                            |    |    |
```

La fonction `f` a des **effets de bord**: elle manipule les variables `i` et `j`. Quelles sont les valeurs stockées dans `i` et `j` à la fin? Quel est le résultat?

En général, les variables sont visibles après leur déclaration.

Souvent les fonctions sont visibles dans toute la portée pour permettre la récursion mutuelle.

```
class Scope {
  Symbol symbol;
  Scope outer;
  Symbol lookup(String name) { ... }
  void enter(Symbol symbol) { ... }
}
```

Projet: pas de `outer`, mais une table globale.

Réalisation

Projet

- Une déclaration valable a comme côté droit un nombre. Par exemple, `a := 3` est une déclaration valable, tandis que `b := a + 1` interdit.
- Une variable utilisée dans un processus doit avoir été déclarée. C-à-d. pas de variables locales.
- Une variable ne doit pas porter le même nom qu'une variable déclarée précédemment.
- Les processus portent des noms distincts.

Les programmes sont généralement petits.

$\Rightarrow$ portée simple: liste linéaire.

Optimisations

Pour réduire le temps pour rechercher et éventuellement insérer des symboles, utilisation des **arbres de recherche** (O(log n) au lieu de O(n)).

Tables de hachage: temps d'accès $\sim$O(1).