Partie V : Analyse des noms

- Les langages de programmation ne sont pas non contextuels
- Règles de contexte pour FuLL
- Représentation des contextes dans un compilateur
- Squelette de spécification des règles de visibilité
- Gestion de la mémoire
- Optimisation
- Affectation

Les langages de programmation ne sont pas non contextuels

- Exemple du compteur : chaque identificateur a besoin d'être déclaré
- « Être déclaré » est une propriété qui dépend du *contexte*.
- En théorie, la syntaxe des langages de programmation peut être entièrement spécifiée dans une grammaire dépendante du contexte.
- Mais en pratique, on définit un *sur-ensemble* non-contextuel du langage en EBNF, et on élimine les programmes illégaux avec d'autres règles.
- Typiquement, ces règles ont besoin d'accéder à la déclaration d'un identificateur.

Règles de contexte pour FuLL

- Pour les identificateurs, FuLL adopte les règles de visibilité standards basées sur la structure en blocs.
- Pour ce qui est de cette discussion, un *bloc* est
  - n'importe quoi entre accolades {}, ou
  - la zone incluant la liste des paramètres et le corps d'une fonction.
- Nous avons alors que :
  - Chaque identificateur a une portée, c-à-d une zone dans le texte du programme à l'intérieur de laquelle on peut s'y référer.
  - La portée d'un identificateur s'étend de l'endroit de sa déclaration jusqu'à la fin du bloc englobant.
  - Il est illégal de se référer à un identificateur en dehors de sa portée.
  - Il est illégal de déclarer deux identificateurs avec le même nom dans le même bloc.
  - Cependant, il est légal de déclarer dans un bloc imbriqué un identificateur qui est aussi déclaré dans un bloc englobant.
  - Dans ce cas la déclaration la plus interne cache la plus externe.

Représentation des contextes dans un compilateur

- On représente les contextes par une structure de données globale, qui stocke pour chaque identificateur visible des informations concernant sa déclaration.
- La structure de données est appelée *table des symboles*, et l'information associée à un identificateur est appelée une *entrée dans la table des symboles* (ou *entrée* pour abréger).
- Comme FuLL a des blocs imbriqués, la table des symboles doit être structurée de la même manière.
- On peut représenter la table des symboles comme une pile de blocs, avec le bloc courant le plus interne au sommet :

```
SymbolTable   = Stack(Block)
Block         = List(Symbol)
Symbol        = ?
```

Symboles

- Un symbole est une structure de donnée qui contient toutes les informations concernant un identificateur déclaré, et que le compilateur doit connaître.
- Les symboles ont un nom et un type.
- Les symboles sont regroupés en portées.
- Il est parfois nécessaire de parcourir tous les symboles d'une portée dans l'ordre de leurs déclarations.

  ⇒ Lier les symboles linéairement avec un champ next.
- Cela amène à la classe suivante pour les symboles.

```
class Symbol {
  Symbol next;
  String name;
  Type type;
  // le constructeur manque
}
```


Types

- Un type est une structure de données qui contient toutes les informations concernant la valeur d'une expression ou d'un symbole (excepté son nom), et que le compilateur doit connaître.
- Les types existent sous des formes diverses : `int`, `boolean`, types liste `list[]`.
- Pour enregistrer les informations concernant les fonctions, on introduit aussi des types fonctionnels. Exemple :

```
list[int] sort(list[int] lst)
```

a le type

```
list[int] (list[int] lst)
```

- Le nom du paramètres `lst` est redondant. On le garde ici car cela peut conduire à une implémentation plus simple si l'on représente les paramètres simplement par la portée qui les contient.

Types (2)

- Cela conduit à la syntaxe abstraite suivante pour les types.

```
Type = IntType
     | BooleanType
     | ListType Type
     | FunType Type Scope
```


Une classe pour les types (1)

- En appliquant systématiquement notre transformation de la syntaxe abstraite vers les classes d'arbre on obtient :

```
class Type {
  static class IntType {}
  static class BooleanType {}
  static class ListType {
    Type elemType;
    ListType(Type elemType) {
      this.elemType = elemType;
    }
  }
  static class FunType {
    Type resType;
    Symbol params;
    FunType(Type resType, Symbol params) {
      this.resType = resType;
      this.params = params;
    }
  }
  static Type intType     = new IntType;
  static Type booleanType = new BooleanType;
}
```

Une classe pour les types (2)

•On peut omettre certaines classes et l'accès peut être optimisé en ajoutant une étiquette (*tag*) qui nous indique la sorte du type.

```
Class Type {
  static final int
    INT     = 1,
    BOOLEAN = 2,
    LIST    = 3,
    FUN     = 4;

  int tag;//une des valeurs ci-dessus

  Type(int Tag) { this.tag = tag }

  static class ListType {
    Type elemType;

    ListType(Type elemType) {
      super(LIST);
      this.elemType = elemType;
    }
  }

  static class FunType {
    Type resType;
    Symbol params;

    FunType(Type resType,
            Symbol params)
    {
      super(FUN);
      this.resType = resType;
      this.params = params;
    }
  }

  static Type
    intType     = new Type(INT),
    booleanType = new Type(BOOLEAN);
}
```


Portées

- Les portées représentent des zones de visibilité.
- Une portée `scope` est une structure de données qui se réfère à tous les identificateurs déclarés à l'intérieur de cette portée.
- Les portées sont imbriquées; il est donc nécessaire d'avoir dans une portée un champ `outer` qui se réfère à la portée directement englobante.
- Cela conduit au morceau de classe suivant.

Une classe pour les portées

```
class Scope {
  Symbol first;
  Scope outer;
  Scope(Scope outer) { this.outer = outer; }
  /** find symbol with given name in this scope.
   *  return null if non exists
   */
  Symbol lookup(String name) {...}
  /** enter given symbol in current scope
   */
  void enter(Symbol symbol) {...}
}
```

- Les portées se réfèrent au premier symbole déclaré dans la portée; on accède aux autres symboles par le champ `next` de la classe `Symbol`.
- Exercice : Écrire des implémentations pour `lookup` et `enter`.

Comment tout cela marche ensemble

- Considérons le programme FuLL

```
int length(list[int] lst) = ...
list[int] sort (list[int] lst) {
  int len = length(lst);
  boolean cond = len < 2;
  if cond then {
    lst
  } else {
    list[int] firstHalf = ... // ****
  }
}
```

- Alors, au point marqué ****, la table des symboles devrait ressembler à ce qui est écrit au tableau.

Gestion de la mémoire

- Les entrées de la table des symboles pour les variables locales des blocs qui ont fini d'être analysés ne sont plus nécessaires.
- Comment s'en débarrasser ?
- En Java, le ramasse-miettes, ou glaneur de cellules (*garbage collector*), s'en occupe.
- En C/C++ la stratégie la plus efficace est un alloueur de mémoire personnalisé qui utilise le marquage (*mark/release*).
- En entrant dans un bloc : marquer le sommet du tas courant
- En sortant du bloc : réinitialiser le sommet du tas à la marque précédente.

Optimisation

- Le schéma courant utilise une recherche linéaire pour les identificateurs.
- Dans un compilateur de production c'est beaucoup trop lent.
- Meilleurs schémas :
  - En plus, lier les entrées comme un arbre binaire et utiliser cela pour la recherche.
  - Utiliser une table de hachage (*hash table*) pour chaque bloc.
  - Utiliser une table de hachage globale (plus rapide).

Spécification des règles de contexte

- Comment les tables de symboles sont-elles utilisées dans un compilateur ?
- Premier chose à se demander : Comment spécifie-t-on l'utilisation des tables des symboles dans les règles de contexte d'un langage ?
- De façon plus générale : Comment spécifie-t-on les règles de contexte ?
- Plusieurs méthodes sont possible.
- Nous utilisons juste une méthode semi-formelle, qui ajoute des *attributs* aux symboles et connecte les attributs au moyen de *contraintes*.

Squelette de spécification des règles de visibilité

```
P     = Program { FD } E
```

"crée une nouvelle portée la plus externe."

```
FD    = FunDecl T(t) name { F } E
```

"traite les paramètres `{F}` dans une portée imbriquée ; crée un symbole dans la portée courante avec le nom `name` donné et un type fonctionnel qui fait référence aux paramètres et type de résultat `t`."

```
VD    = ValDecl T(t) [ name ] E
```

"crée un nouveau symbole dans la portée courante avec un nom `name` et type `t` donnés."

```
T(t)  = int                    t = Type.intType
      | boolean                t = Type.booleanType
      | list[T(t1)]            t = new Type.ListType(t1)
E     = Ident(name)            e = findSymbol(name)
```

Grammaires attribuées

- Une syntaxe dépendant du contexte est parfois spécifiée en utilisant une *grammaire attribuée*.
- Similaire à ce que l'on a fait, mais complètement formel.
- Les grammaires attribuées reposent sur la syntaxe non contextuelle concrète.
- On donne des attributs aux symboles, pouvant avoir un type quelconque.
- Les attributs sont évalués par des affectations similaires à nos contraintes.
- On représente les attributs comme des variables d'instance des nœuds de l'arbre.

Système de types

- Exprimer une syntaxe dépendant du contexte comme un système déductif.
- Les jugements sont de la forme $E \vdash$ <terme> : <type>.
- Un programme P est bien typé ssi un jugement $E \vdash P: T$ est prouvable.
- Exemple : Une règle de typage pour l'addition :

$$\frac{E \vdash A: \text{int} \quad E \vdash B: \text{int}}{E \vdash A + B: \text{int}}$$

- Généralement on garde aussi un environnement E qui représente la table de symbole courante dans un jugement.
- Les systèmes de type sont souvent plus concis et lisibles que les grammaires attribuées.
- Les grammaires attribuées sont plus proches d'une implementation.

Structures de données pour consultation efficace de la table des symboles

- Avec la solution précédente, trouver un symbole prenait O(N) étapes, où N est le nombre de symboles visibles au point où le symbole est consulté.
- Si N croît linéairement avec la longueur du programme, le temps total nécessaire pour accéder aux symboles croît de façon quadratique.
- Cela peut être amélioré en maintenant des structures de données qui accélèrent la consultation des symboles.
- Deux solutions classiques :
  - *les arbres de recherche*
  - *les tables de hachage*
- Implémentées ici comme extensions de la classe Scope.

Arbres de recherche binaires

- Un arbre de recherche binaire peut être utilisé si les clés sont totalement ordonnées.
- Un nœud dans l'arbre de recherche contient une entrée de la table ainsi que des pointeurs `left` et `right` pour les sous-arbres.
- Invariant pour chaque nœud n:
  - toutes les entrées rangées dans le sous-arbre gauche sont plus petites que l'entrée rangée dans n , et
  - toutes les entrées rangées dans le sous-arbre droit sont plus grandes que l'entrée rangée dans n.
- Donc, la consultation nécessite de rechercher dans au plus un sous-arbre.

Une classe Scope avec arbre de recherche

```
class TreeScope extends Scope {

  static class SymTree {
    SymTree left = null, right = null;
    Symbol sym;
    SymTree (Symbol sym) {
      this.sym = sym;
    }
  }

  SymTree root = null;

  SymTree insert(SymTree t, Symbol sym){
    if (t == null)
      return new SymTree(sym);
    else {
      int rel =
        sym.name.compareTo(t.sym.name);
      if (rel < 0)
        t.left = insert(t.left, sym);
      else if (rel > 0)
        t.right = insert(t.right, sym);
      return t;
    }
  }

  Symbol find(SymTree t, String name) {
    ...
  }

  Symbol lookup(String name) {
    return find(root, name);
  }

  Symbol enter(Symbol sym) {
    root = insert(root, sym);
  }
}
```


Suppressions

- Les suppressions dans un arbre de recherche binaire sont un peu plus difficiles.
- Soit n le nœud à supprimer. Par quoi doit être remplacé n ?
- Cas facile : un sous-arbre de n est nul. Prendre l'autre.
- Cas plus difficile : aucun sous-arbre n'est nul. Il faut trouver une nouvelle racine.
- Solution : Prendre le plus petit nœud du sous-arbre droit (on pourrait prendre aussi le plus grand nœud du sous-arbre gauche).
- Comment trouver ce plus petit nœud ?

La méthode Delete

```
SymTree delete(SymTree t, Symbol sym) {
  if (t == null)
    return t;
  else {
    int rel =
      sym.name.compareTo(t.sym.name);
    if (rel < 0)
      t.left = delete(t.left, sym);
    else if (rel >= 0)
      t.right = delete(t.right, sym);
    else {
      if (t.right == null)
        t = t.left;
      else if (t.left == null)
        t = t.right;
      else {
        SymTree prev = null;
        SymTree newt = t.right;
        while (newt.left != null) {
          prev = newt;
          newt = newt.left;
        }
        if (prev == null)
          t.right = newt.right;
        else
          prev.left = newt.right;
        newt.left = t.left;
        newt.right = t.right;
        t = newt;
      }
    }
    return t;
  }
}
```


Complexité des opérations sur la table

- Coût pour

	en moyenne	au pire
consulter	O(log N)	O(N)
insérer	O(log N)	O(N)
supprimer	O(log N)	O(N)

- On peut obtenir un meilleur comportement dans le pire des cas en équilibrant l'arbre.
- Pour des insertions aléatoires, le nombre de comparaisons est environ 40% plus important pour un arbre non équilibré que le log(N) pour un arbre parfaitement équilibré.

Solutions à base de tables de hachage

- Une table de hachage est similaire à un tableau indexé par les entrées de la table (→ temps d'accès constant).
- Il faut tout d'abord associer des entiers aux entrées de la table.
- C'est le rôle de la méthode

  `int hashCode()`

  dans la classe `Object`.
- Il reste deux problèmes :
  - intervalle d'indices trop grand (typiquement 28 à 32 bits)

    ⇒ utiliser le code de hachage modulo la taille du tableau désiré.
  - des collisions sont possibles

    ⇒ les éléments du tableau sont des listes linéaires d'entrées avec le même code de hachage.

Code avec table de hachage

```
class HashScope extends Scope {

  static class Entry {
    Entry next;
    Symbol sym;

    Entry (Entry next, Symbol sym) {
      this.next = next;
      this.sym = sym;
    }
  }

  final int N = 1024;
  Entry [] elems = new Entry [N];

  Symbol enter(Symbol sym) {
    int i = sym.name.hashCode() % N;
    elems[i] = new Entry (elems[i], sym);
  }

  Symbol lookup(String name) {
    ...
  }
```

Code avec table de hachage (2):

```
  void deleteAll(Symbol sym) {
    if (sym != null) {
      deleteAll(sym.next);
      int i = sym.name.hashCode() % N;
      elems[i] = elems[i].next;
    }
  }

  void leave() {
    deleteAll(first);
  }
}
```

- Remarques :
  - On utilise uniquement une table pour toutes les portées.
  - Donc, l'initialisation et la consultation des portées est plus efficace.
  - D'un autre côté, on doit alors supprimer les entrées de la table en sortant d'une portée.

Ensembles et tables

- Généralisation :

```
class Table<K,D> {
  D get (K key);
  void put (K key, D data);
  void delete (K key);
  Iterator<K> elements();
}
```

- Cas spécial :

```
class Set<A> {
  Boolean contains (A elem);
  void enter (A elem);
  void delete (K key);
  Iterator<A> elements();
}
```

`K, D, A` sont des paramètres de type

Utilisation:

```
Table<String, Symbol> scope = new Table<String, Symbol>
. . .
Symbol sym = scope.get(name)
```

Voir:

C++ Templates
Ada/Eiffel Generics
GJ/Pizza Generics

Paramètres de type

- La version courant de Java ne permet pas d'écrire des paramètres de type.
- À la place, on utilise la classe Object. E.g.

```
class HashTable {
  Object get (Object key);
  void put (Object key, Object data);
  void remove (Object key);
  Iterator elements();
}
```

- L'utilisateur doit insérer des « type casts » pour accéder aux éléments.

```
Symbol sym = (Symbol)table.get(name)
```

- Désavantage : manque de documentation, sûreté.
- De meilleures solutions avec C++ (templates), ou dans de possibles prochaines versions de Java (GJ) ou C#.