Analyse sémantique
Gestion de symboles

michel.chilowicz@univ-mlv.fr

Squelette de tatou à 9 bandes (par ellenm1, CC By)

Catégories de symboles

- Types non primitifs
  - Structure, classe
- Méthodes, fonctions
- Constantes
- Variables
  - De portée globale (visibles d'unités externes)
    - Champs d'unités de compilation
    - Champs de structures, classes
  - De portée locale (visibles à l'intérieur d'une fonction)
    - Arguments de fonction
    - Variables déclarées dans le corps de la fonction

Accès aux symboles

- Chaque symbole a une visibilité plus ou moins large.
- La visibilité par défaut peut parfois être modifiée par des mots-clés (sur des types, fonctions et variables) :
  - En Java : *public*, *protected*, *private*
  - En C++ : sections *public*, *protected*, *private* et relations d'amitié (*friend*)
- Une variable peut être protégée en réécriture (final en Java, const en C(++)...)

Utilité de la table de symboles

- Vérification de l'existence d'un symbole, liaison des sites d'utilisation d'un symbole à sa déclaration (résolution d'appels de fonctions)
- Vérification du respect de la politique de visibilité
- Détermination du type d'un symbole et de la cohérence des types

Réalisation de ces tâches durant la compilation... et/ou lors de l'exécution (langages dynamiques de script)

Exemples d'utilisation de tables de symboles

```
public class FibonacciNumberGenerator
{
	private final int fib0 ; private final int fib1 ;
	private int fib2 ; // Le champ privé fib2 n'est jamais accédé
	private int lastComputed ;
	private int[] firstNumbers ;

	public FibonacciNumberGenerator(int fib0, int fib1)
	{
		this.fib0 = fib0 ;
		// On a oublié d'initialiser le champ final fib1 : erreur
		firstNumbers[0] = fib0 ; firstNumbers[1] = fib1 ;
		fib2 = fib0 + fib1 ;
		// firstNumbers risque d'être non initialisé
		lastComputed = 1 ;
	}

	public int get(int k) // Cette méthode privée n'est jamais accédée
	{
		for (int i = lastC + 1; i <= k ; i++) // La variable lastC n'est pas définie
			firstNumbers[i] = firstNumbers[i-1] + firstNumbers[i-2] ;
		return firstNumbers[k] ;
	}

	public long[] getFirstNumbers() { return (long[])firstNumbers ; /* Cast invalide */ }
}
```

Quand et comment typer un symbole ?

- Typage statique
  - Explicite (C(++), Java...)
    - Ex : déclaration de variable locale int c = 7 ; (en C ou Java)
  - Implicite (Haskell, Ocaml...) : inférence de type à la compilation
    - Ex en Caml : let f = function x,y -> (x + y)/2 ;;
      val f : int * int -> int = <fun>
  - Mixte (Scala) : inférence avec indices de types
- Typage dynamique : une variable peut avoir différents types selon le contexte d'exécution (Python, Erlang, PHP, JavaScript...)
  - Ex. en Python : lambda x,y : (x + y)/2
    x et y sont des entiers ? des flottants ? un autre type dont on aurait surchargé + ?

Gérer les symboles avec un visiteur

- Un unique parcours en profondeur de l'AST est-il suffisant ?
  - Pour certains (vieux) langages tel que C : oui
    - Pré-déclaration de types (structures, fonctions)
  - Dans le cas général (Java, C#...) : non
    - Dépendances récursives :
      - class Tree { int root ; Forest children ; }
      - class Forest { List<Tree> trees ; }
    - Ordonnancement quelconque des fonctions :
      - int g(int x) { return f(x * x) ; }
      - int f(int y) { return 2 * y ; }

Imports

- Une unité de compilation peut dépendre d'autres unités : dépendances matérialisées par des imports
- Avant de générer sa table de symboles, il faut générer la table de symboles des unités importées qui sont visibles de cette unité.

Portée des symboles

- Chaque structure de code (unité de compilation, classe, méthode, structure de code...) représente un *scope*
- Selon les langages
  - Surcharge impossible des symboles
    - Erreur reportée lors de la génération de la table de symboles
  - Surcharge possible
    - Avec possibilité d'accéder à l'identificateur surchargé...
    - … ou pas
- Utilisation d'une pile de tables de symboles : recherche d'un symbole du scope le plus local au plus global

Scopes en Java

```
public class Rectangle
{
	public class InnerPoint
	{
		int x, int y ;

		public InnerPoint(int x, int y) { this.x = x ; this.y = y ; }

		public int getAbsoluteX() { return Rectangle.this.x + this.x ; }
	}

	int x, y, width, height ;

	public Rectangle(int x, int y, int width, int height) { … }

	// We enumerate all the points inside a rectangle
	public List<InnerPoint> getInnerPoints(int number)
	{
		…
	}
}
```

Surcharge des champs *x* et *y* de *Rectangle* par des champs de même nom dans *InnerPoint*

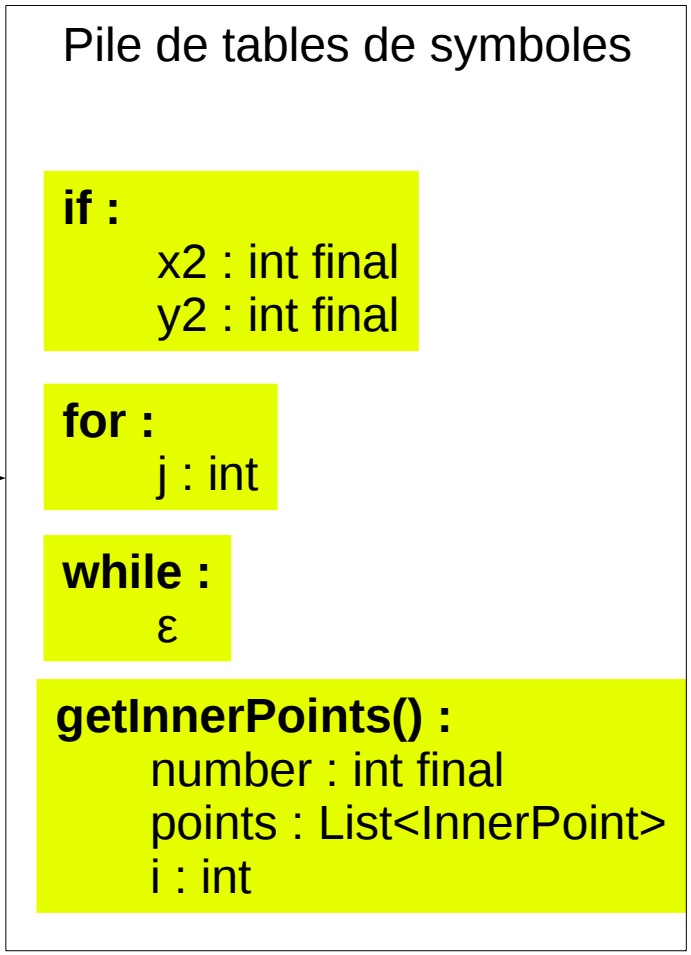
Surcharge des champ *x* et *y* de *InnerPoint* par les variables locales *x* et *y*

Rectangle
this : Rectangle
x : int
y : int
width : int
height : int
getInnerPoints(int)

InnerPoint
this : InnerPoint
Rectangle.this : Rectangle
x : int
y : int
getAbsoluteX()

Scopes en Java (suite)

```
public List<InnerPoint> getInnerPoints(final int number)
{
        List<InnerPoint> points =
              new ArrayList<InnerPoint>(number) ;
        int i = 0 ;
        while (i < width)
        {
              for (int j = 0; j < height ; j++)
                    if (points.size() < number)
                    {
                          final int x2 = x + i ;
                          final int y2 = y + j ;
                          points.add(new InnerPoint(x2, y2)) ;
                    }
              i++ ;
        }
        return points ;
}
```



Clôtures de fonctions

- Certains langages permettent d'imbriquer des fonctions
  - Une fonction interne peut accéder aux variables locales de la fonction englobante

```
# En Python
def f(x) :
    def g(y) :
        return x + y
    return g
f(1)(2) # Retourne 3
```

```
// En Java
public static Computer createComputer(final int x)
{
    return new Computer()
    {
        public int compute(int y) { return x + y ; }
    }
}
```

Collection et résolution de symboles

1. Détermination des symboles des unités externes référencées
2. Détermination des symboles globaux de l'unité courante : variables, déclaration de fonctions
3. Génération des table de symboles des fonctions
   Résolution des symboles référencés :
   - Recherche des variables du *scope* le plus local au plus global
   - Liaison des appels de fonction

Liaison des appels de fonctions

- Langages sans surcharge de fonctions (C, Pascal...) : liaison simple
- Langages avec surcharge et redéfinition de fonctions : quelle signature de fonction choisir ?
  - Exemple en Java :
    - String f(Integer i, Object s) { return i + « : » + s ; }
    - String f(Object i, String s) { return i + « , » + s ; }
    - String g() { return f(0, « bonjour ») ;
      /* Quelle fonction f appeler ? */ }