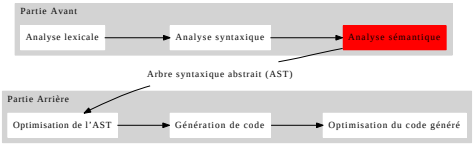
Compilateurs : Analyse sémantique

Matthieu Amiguet

2007 – 2008

haute école arc ingénierie
neuchâtel berne jura saint-imier le locle delémont

Vous êtes ici 2



Analyse sémantique 3

- À la sortie de l'analyse syntaxique, on se trouve avec un arbre syntaxique (abstrait) qui contient toutes les informations pertinentes sur la *structure syntaxique* du programme
  - On a donc vérifié qu'il est syntaxiquement correct...
  - ... ce qui ne veut pas dire qu'il est *sémantiquement* correct !
  - De même, on a compris sa *structure* syntaxique...
  - ... ce qui ne veut pas dire qu'on a compris sa *signification* !

Remarque 4

Normalement...

Syntaxe *Forme* [d'un programme donné]

Sémantique *Signification* [de ce programme]

Dans le contexte des compilateurs

Analyse syntaxique Analyse de tout ce qui peut s'analyser sans prise en compte du contexte

Analyse sémantique Analyse de ce qui nécessite le contexte.

Cohérence à distance 6

```
float f(int x, int y);
int i;

[...]

i = 12.3;
i = f(12);
i = f(1.2, 2.3);
```


Étudier le flux de contrôle 7

```
class DoesntCompile {

    static int test(int i) {
        int result;
        if (i>0) result=0;
        if (i<=0) result=-1;
        return result;
    }

    static public void main(String[] argv) {
        test(1);
    }

}
```

⇝ `DoesntCompile.java:7: variable result might not have been initialized`

Ce qu'on *pourrait* vérifier 8

En plus des exemples ci-dessus...

- Les identificateurs ne sont-ils définis qu'une seule fois (par portée) ?
- Toute fonction contient-elle un *return* ?
- Y a-t-il du code inatteignable ?
- ...

Que *doit-on* vérifier ? 9

- Question de choix
- Moins de vérification
  - Plus de liberté de programmation
  - Moins de travail pour le compilateur
  - Plus de risques de problèmes à l'exécution
  - Exemple : C
- Plus de vérification
  - Programmation plus contrainte
  - Plus de travail pour le compilateur
  - Moins de risques à l'exécution
  - Exemple : Java.

Portée 10

```
i = 1

def f(i):
   def g():
      i = 3
      print i
   i =2
   g()
   print i

f(i)
print i
```


Polymorphisme 11

```
class Carre extends Forme{
   [...]
   float aire() {return cote * cote; }
}

class Cercle extends Forme{
   [...]
   float aire() {return pi * rayon * rayon; }
}

Forme myForme;
[...]
a = myForme.aire();
```


Polymorphisme – suite 12

```
class Toto {
   public Toto(int i) {
       [...]
   }

   public Toto(int i, int j) {
       [...]
   }
}

[...]

Toto t = new Toto(1,2);
```

Portée des identificateurs 14

- La plupart des langages de programmation permettent au même identificateur d'être utilisé en différents endroits du programme sans forcément se référer au même objet
- On appelle *portée* d'un identificateur la zone du programme où cet identificateur est visible
- Il existe deux mécanismes de portée
  - la portée *statique* peut être déterminée sur la base de l'arbre syntaxique (et peut donc être calculée lors de l'analyse sémantique)
  - la portée *dynamique* est déterminée à l'exécution.

Portée statique 15

- Implémentée dans la plupart des langages de programmation modernes (C, Java, Pascal, . . .)
- Parcours en profondeur de l'*arbre syntaxique*
  - on maintient une pile des portées dans laquelle on place les identificateurs visibles dans la portée
  - pour chercher un identificateur, on cherche depuis le haut de la pile
  - on "annote" chaque noeud contenant une variable pour lever l'ambiguïté.

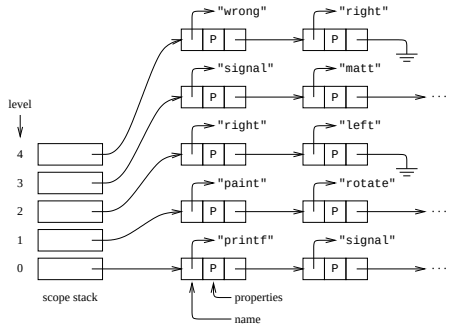
Portée statique – exemple 16

```
void rotate(double angle) {
    ...
}
void paint(int left, int right) {
    Shade matt, signal;

    ...
    {   Counter right, wrong;
        ...
    }
}
```

- On peut déterminer à quel emplacement mémoire se réfère chaque identificateur *statiquement* sur la base de l'arbre syntaxique.

Portée statique – exemple 17



Portée dynamique 18

- Tend à être moins utilisée
  - Perl, TEX, Postscript, certains dialectes Lisp, . . .
- Un identificateur se rapporte à sa dernière déclaration *dans le flux de contrôle*
  - Ne peut donc être déterminé qu'à l'exécution !
  - Le mécanisme de gestion est similaire, mais la pile des portées est gérée à l'exécution plutôt qu'au moment de l'analyse sémantique
- Considéré comme difficile à maîtriser et source de beaucoup d'erreurs
- Le principal argument en faveur de la portée dynamique est la facilité de faire des routines *adaptables*.

Portée dynamique – exemple 19

```
a : integer

procedure first
     a := 1
procedure second
     a : integer
     first()

a := 2
if read_integer() > 0
     second()
else
     first()
write_integer(a)
```

Vérifications statiques 21

- Toutes les propriétés qui ne dépendent que de l'arbre syntaxique lui-même sont assez faciles à traiter
  - Type des affectations, portée statique, ...
- Il suffit de faire un parcours récursif de l'arbre et, pour chaque noeud, effectuer les vérifications correspondantes
  - Si le type du noeud est une affectation, vérifier que le type à gauche est le même que le type à droite...
- Par contre, les propriétés qui dépendent du flot de contrôle nécessitent un peu plus de travail...

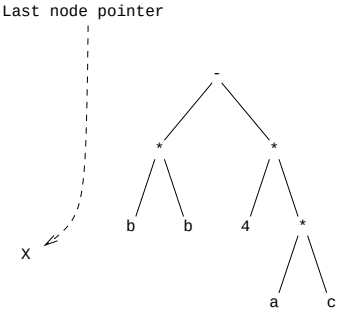
"Couture" de l'arbre abstrait 22

- On peut essayer de relier les noeuds de l'arbre abstrait dans l'ordre où ils vont être rencontrés par le programme
- On a donc une deuxième structure sur les noeuds qui vient se *superposer* à l'arbre
- On appelle cette construction *couture* de l'arbre abstrait.

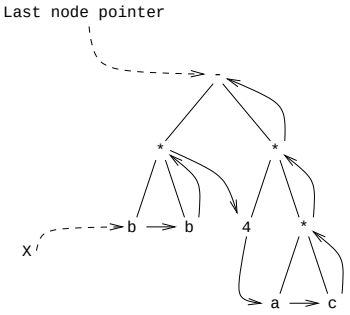
Comment coudre ? 23

- Le but est que chaque noeud possède un pointeur sur le(s) noeud(s) qui le sui(ven)t dans le flot de contrôle
- Pour cela, on a besoin d'un sous-programme par type de noeud, qui
  - prend en paramètre le noeud à traiter
  - appelle le sous-programme de couture de ses enfants
- On obtient donc un parcours récursif de l'arbre
- On doit garder dans une variable globale un pointeur sur le dernier noeud cousu jusqu'à présent.

Exemple – situation initiale 24



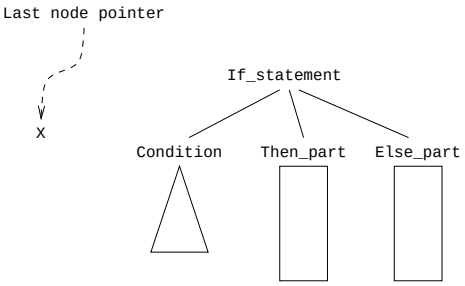
Exemple – situation finale



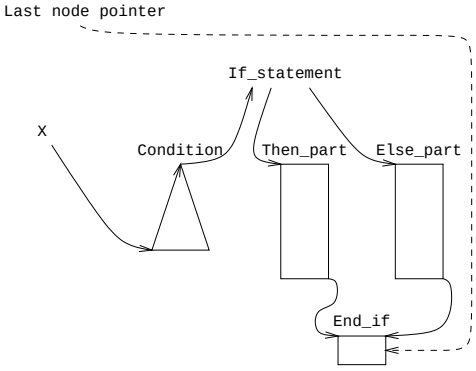
Et en cas de bifurcation ?

- Certains noeuds peuvent avoir plusieurs successeurs potentiels
  - enoncés conditionnels
- Il faudra donc qu'un noeud ait plusieurs sucesseurs (bifurcation)...
- ... et parfois qu'un noeud ait plusieurs prédecesseurs (réunion)
  - Il est parfois plus pratique de rajouter un noeud spécial pour cela
  - Ce noeud fait donc partie de l'arbre cousu, mais pas de l'arbre abstrait.

Exemple – situation initiale



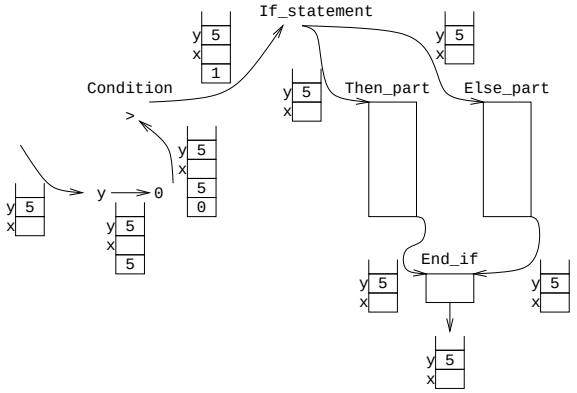
Exemple – situation finale



L'idée

- On a maintenant une représentation du flot de contrôle du programme
- On peut *simuler* des chemins d'exécution pour en étudier les caractéristiques
- On choisit un chemin
  - On attache à chaque arc du graphe de contrôle des informations sur le contexte d'exécution à ce point (représentation de la "pile d'exécution")
  - Cette représentation peut être une approximation plus ou moins précise selon les besoins
- On choisit ensuite un autre chemin et on recommence...
  - Cela permet de vérifier les initialisations de variables, les portions de code inaccessibles, etc.

Exemple



Ça peut se compliquer

- Lorsque le flot de contrôle devient trop compliqué...
  - beaucoup de conditionnelles
  - boucles
  - ...

  ... il n'est plus possible de faire un parcours "naïf"
  - explosion combinatoire !
- On essaie donc de dégager des propriétés communes aux différents passages dans les noeuds (en gros)
  - Les algorithmes *ad hoc* deviennent compliqués...

Automatisation de l'analyse sémantique 33

- La phase d'analyse sémantique est moins systématiquement automatisée que les deux précédentes
- Les méthodes utilisées sont plus jeunes
  - L'interprétation symbolique peut être automatisée par une méthode appelée *équation de flots de données*
  - Il reste par contre à coudre l'arbre
- Une autre méthode repose sur l'utilisation de *grammaires attribuées*
  - Il n'existe pourtant pas (encore ?) d'outil aussi standard que lex/yacc dans ce domaine...
- Nous ne verrons pas en détail les techniques d'automatisation de l'analyse sémantique