

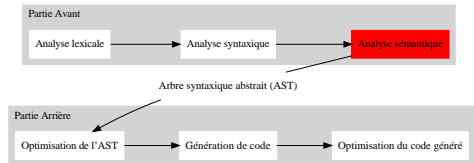
Compilateurs : Analyse sémantique

Matthieu Amiguet

2007 – 2008



Vous êtes ici



Analyse sémantique

- 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*!

- 1 Analyse sémantique
- 2 Portée
- 3 Méthodes manuelles
- 4 Automatisation

Syntaxe et sémantique

5

Rappel

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);
```

Portée

7

```
i = 1

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

f(i)
print i
```

Polymorphisme

8

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

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

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

Flux de contrôle

9

```

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);
    }
}

```

Ce qu'on pourrait à vérifier

11

- Le type des affectations est-il correct ?
- Les identificateurs ne sont-ils définis qu'une seule fois (par portée) ?
- Le nombre de paramètres d'un appel de fonction correspond-il à la définition de cette fonction ?
- Toute fonction contient-elle un *return* ?
- Y a-t-il du code inatteignable ?
 - Optimisation
 - Avertissement
- Quelle est la "durée de vie" des variables ?
 - Attribution des registres
- Les variables sont-elles toujours initialisées avant leur utilisation ?
- Peut-on pré-calculer des valeurs ?
- ...

Levée des ambiguïtés

10

- Lier chaque variable à l'emplacement mémoire correspondant
 - portée
 - *namespaces*
- Lier chaque appel de fonction/méthode au bon code en mémoire
 - surcharge
 - polymorphisme
 - *namespaces*
- ...

Que doit-on vérifier ?

12

- 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.

- Certaines ambiguïtés/vérifications se contentent de l'arbre syntaxique et des tables de symboles
 - Signature des sous-programmes
 - Types des affectations
 - ...
- Par contre, certaines vérifications dépendent du *flot de contrôle* dans le programme
 - Initialisation des variables
 - Code inatteignable
 - Tout chemin dans une fonction contient-il un `return` ?
 - ...

- 1 Analyse sémantique
- 2 **Portée**
- 3 Méthodes manuelles
- 4 Automatisation

- 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.

- 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 "annotate" chaque noeud contenant une variable pour lever l'ambiguïté.

Portée statique – exemple

17

```

void rotate(double angle) {
    ...
}

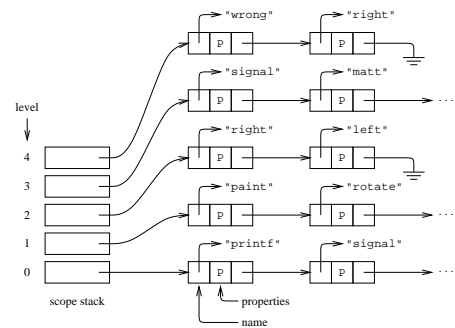
void paint(int left, int right) {
    Shade matt, signal;
    ...
    { Counter right, wrong;
        ...
    }
}

```

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

Portée statique – exemple

18



Portée dynamique

19

- Tend à être moins utilisée
 - Perl, TeX, Postscript, certains dialectes Lisp, ...
- Une 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

20

```

a : integer

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

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

```

1 Analyse sémantique

2 Portée

3 Méthodes manuelles

- Étude du flot de contrôle
- Interprétation symbolique

4 Automatisation

Analyse sémantique
Méthodes manuelles
Étude du flot de contrôle

“Couture” de l’arbre abstrait 23

- 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.

Analyse sémantique
Méthodes manuelles

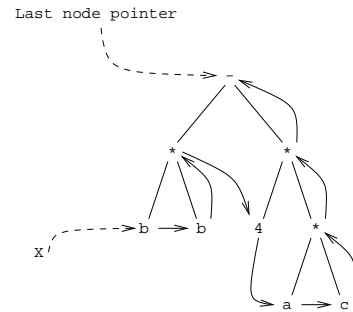
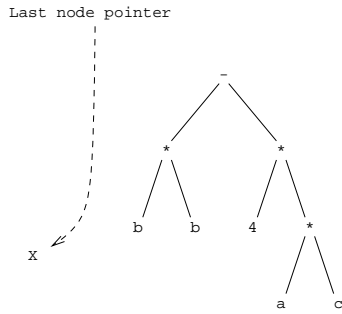
Vérifications statiques 22

- 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. . .

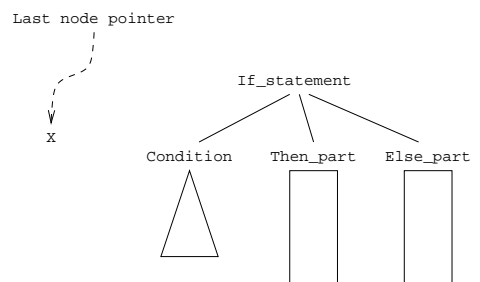
Analyse sémantique
Méthodes manuelles
Étude du flot de contrôle

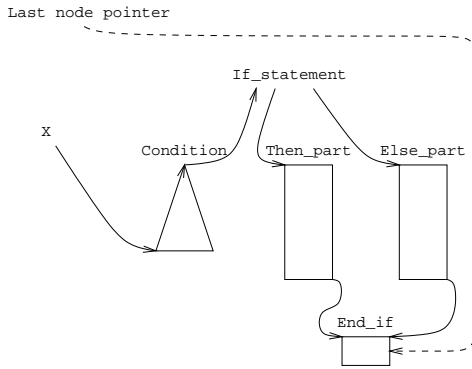
Comment coudre ? 24

- 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.

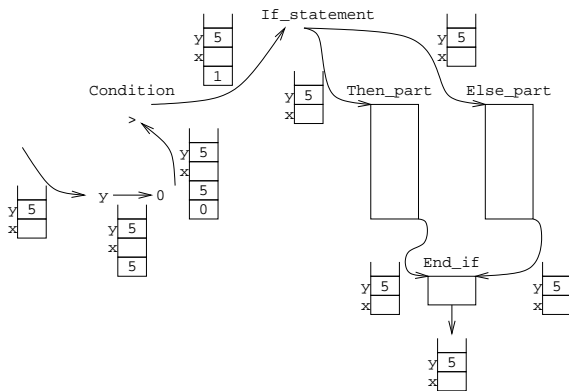


- Certains noeuds peuvent avoir plusieurs successeurs potentiels
 - énoncés conditionnels
- Il faudra donc qu'un noeud ait plusieurs successeurs (bifurcation)...
- ... et parfois qu'un noeud ait plusieurs prédecesseurs (réunion)
 - Il est parfois plus pratique de rajouter un noeud spécial pour cela
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





- 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.



- 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...

- 1 Analyse sémantique
- 2 Portée
- 3 Méthodes manuelles
- 4 Automatisation

- 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
