Compilation

Vb. Analyse sémantique statique
Analyse de nom et analyse de type

Jacques Farré

Jacques.Farre@unice.fr

Introduction

- les langages contiennent des *entités* de différentes natures
  - types, variables, fonctions ...
  - comment les représenter ?
    - une classe (ou une structure) pour chaque nature d'entité
    - les exemples seront donnés tantôt en Java, tantôt en C

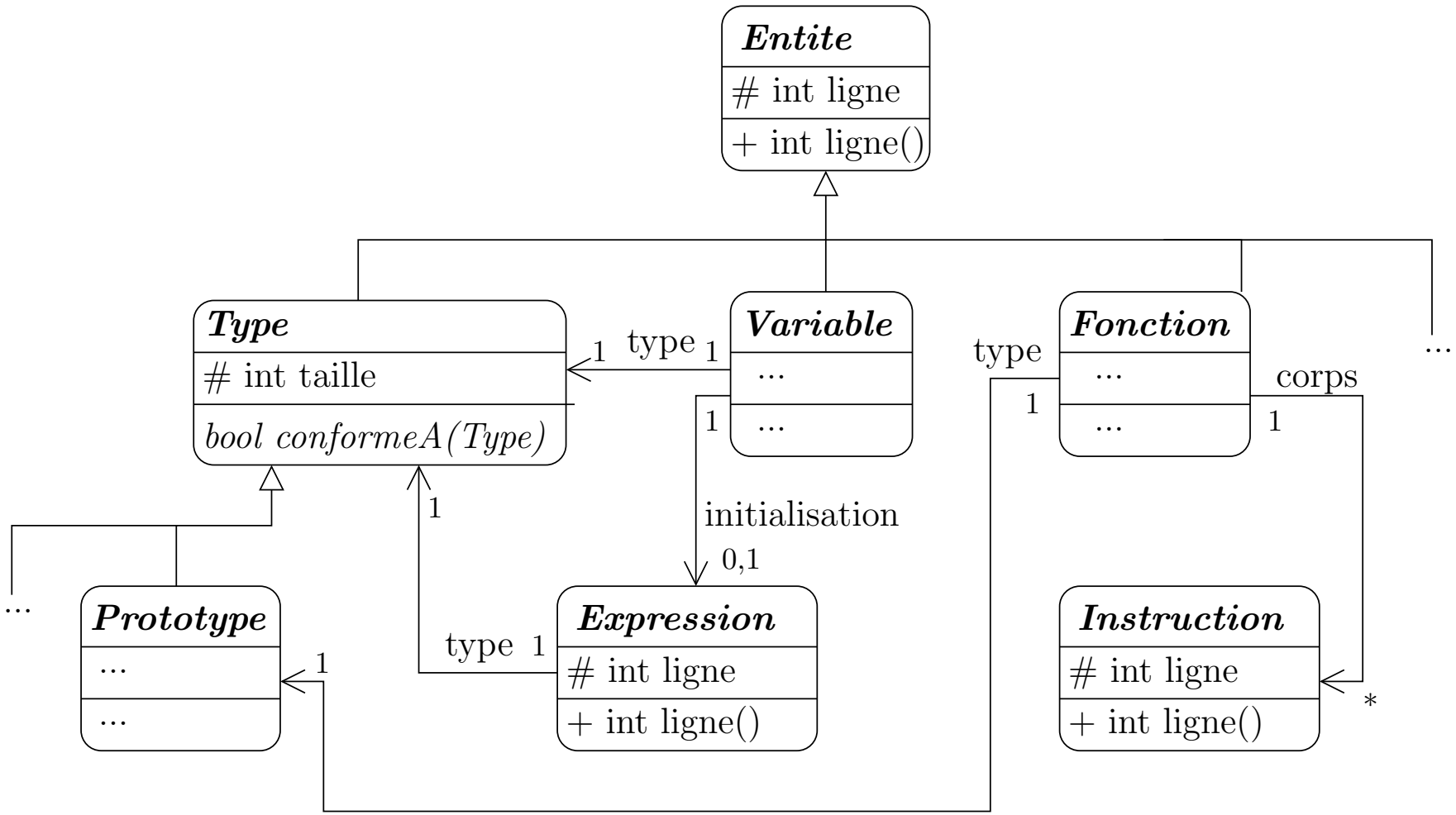
- analyse de nom : traitement des déclarations et des utilisations d'identificateurs
  - notion de *portée* et de *visibilité* d'un identificateur
  - nécessite une *table des symboles*

- analyse de type : notion d'équivalence de type, validité des types des opérandes, surcharge, conversion ...
  - ne pas confondre les types qu'on veut représenter, et les types des structures qu'on utilise pour cette représentation

Représentation des différentes *entités*

- les types
  - types simples : *self*-définis, avec le plus souvent des règles de conversion et de compatibilité (entier $\rightarrow$ réel ...)
  - types structuré : basés sur d'autres types (et finalement sur les types simples)
    - par exemple, tableau : type des éléments (et éventuellement des indices)
    - ou fonction : prototype (type des paramètres et du résultat)
    - ou encore classes : les attributs, les méthodes, les liens d'héritage ...
  - pour la production de code, il faut connaître la taille des instances d'un type donné sur la machine cible
- les autres entités ont toutes un type, notamment
  - variables : le type suffit pour l'analyse statique (mais il faut une information d'allocation et/ou une adresse mémoire pour la production de code)
  - fonctions (et méthodes) : au moins leurs paramètres/variables locales et arbre abstrait de leur corps

Les entités dans une représentation objet



Les entités dans une représentation non objet

les différents types

- on peut s'inspirer du diagramme de classes pour créer une structure polymorphe, par exemple pour les différents genres de types

```
typedef struct Type* PType;

typedef struct Tableau {
  PType elem; /* le type des éléments du tableau */
} Tableau;

typedef struct Classe {
  ListeEntites attr; /* la liste des attributs */
  ListeEntites meth; /* la liste des méthodes */
} Classe;

typedef struct Prototype {
  PType result;          /* le type du résultat */
  ListeEntites param;  /* les paramètres */
} Prototype;
```

Les entités dans une représentation non objet

types comme sous-classe d'entité

```
typedef enum { typeSimple, typeTableau, typeFonction, typeClasse } GenreType;

typedef struct Type {
  int taille; GenreType genre;
  union {
    Tableau   tab;
    Classe    cla;
    Prototype pro;
  };
} Type;

typedef enum { unType, uneVariable, uneFonction } NatureEntite;

typedef struct Entite {
  int ligne; NatureEntite nature;
  union {
    Type     typ;
    Variable var;
    Fonction fct;
  };
} Entite;
```

Portée d'un identificateur

- la portée d'un identificateur est la portion d'un programme dans lequel il peut être utilisé
  - par exemple, pour { `... int i ; ...` }, la portée de `i` va de sa déclaration à la fin du bloc
  - mais pour un attribut, sa portée est toute les méthodes de la classe (et des classes héritières) indépendamment de l'endroit où il est déclaré (donc, sa portée comprend des méthodes déclarées avant lui)
- la visibilité est la portion de la portée où l'identificateur n'est pas masqué par un autre (si le langage autorise ce masquage)
  - par exemple `... int i ; ... for (int i = ...) ...` : le second `i` masque le premier dans le corps de la boucle
- ne pas confondre avec le contrôle d'accès (public, privé ...)

Représentation d'une portée

- fondamentalement, une portée est une association entre un identificateur et l'entité qu'il dénote, association établie pour une partie du programme à analyser
- une association peut être implémentée de diverses façons
  - une liste de couples *(identificateur, entité)*, liste ordonnée (sur les identificateurs) de préférence
  - un arbre binaire de recherche ou un B-arbre
  - une table d'adressage dispersé
  - une table d'accès direct si on peut numéroter les identificateurs
- les constructions dont l'implémentation doit posséder une portée sont celles dans lesquelles on peut déclarer des identificateurs de portée locale à cette construction, donc
  - structures/unions/classes (champs/attributs)
  - fonctions/méthodes (paramètres et autres objets locaux)
  - blocs $\{\ldots\}$ et éventuellement les boucles (uniquement des variables en général)
  - unités de compilation (portée *globale*), modules, paquetages ...

Rappel sur les tables à adressage dispersé

- une fonction $H(s)$ associe une valeur entière dans $[1 : N]$ (ou $[0 : N-1]$ à une chaîne, par exemple la fonction *x65537* :

```
unsigned H (char* p) { /*
  unsigned h = 0;
  while (*p) h = h * 65537 + *p++;
  return h % 257 + 1; /* --> [1..257] */
}
```

- si $H(s) = H(s')$, collision
  - soit chercher/ranger s' à une autre place dans la table (table fermée)
  - soit gérer une liste de collisions (table ouverte)

Table ouverte et table fermée

- *table ouverte* : chaque entrée de la table est une sous-table (une liste simplement chaînée ou un arbre binaire par exemple), contenant toutes les chaînes en collision pour une même valeur
- *table fermée* : en cas de collision, on applique une fonction H' pour trouver une autre valeur de hash-code, jusqu'à trouver une position libre ou avoir épuisé toutes les possibilités :
  - la fonction H' la plus simple engendre la suite $h_k = (h_{k-1} + 1) \bmod N$ (table saturée quand $h_k = h_0$).
  - une meilleure fonction, car elle disperse plus dans la table les chaînes en collision, engendre la suite $h_k = (h_{k-1} + h_0) \bmod N$ (il faut avoir en ce cas $1 \leq h_0 \leq N - 1$) ;
    si N est premier, cette suite permet de parcourir toutes les entrées de la table entre 1 et $N - 1$ (selon des pas de h_0), quelle que soit la valeur de h_0 ; la table est saturée quand k atteint N (car $h_N = 0$).

Exemple de table ouverte et de table fermée

- pour $H(titi) = H(chose) = h$ et $H(bidule) = H(machin) = g = 2h$

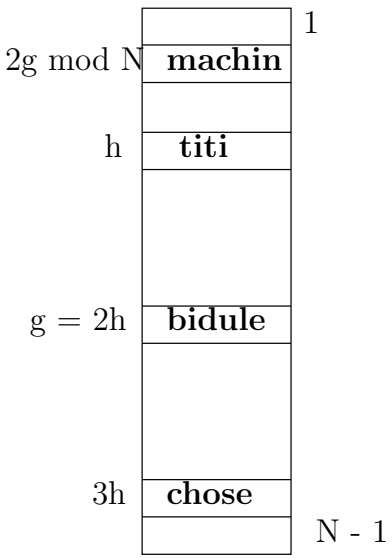


table fermée

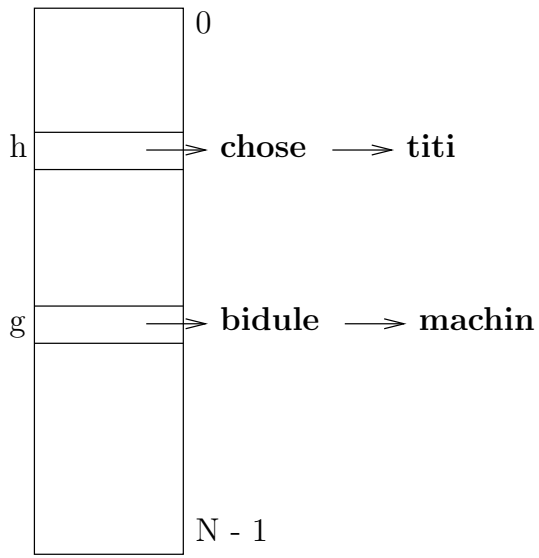


table ouverte

- l'avantage d'une table fermée est qu'elle ne consomme pas de mémoire pour représenter les éléments des listes de collisions et qu'elle fait correspondre un entier unique à chaque chaîne
- son inconvénient est évidemment que le nombre total de chaînes est limité ; elle ne peut donc être utilisée que lorsque le nombre de chaînes est connu, ou au moins borné

Emboîtement de portées

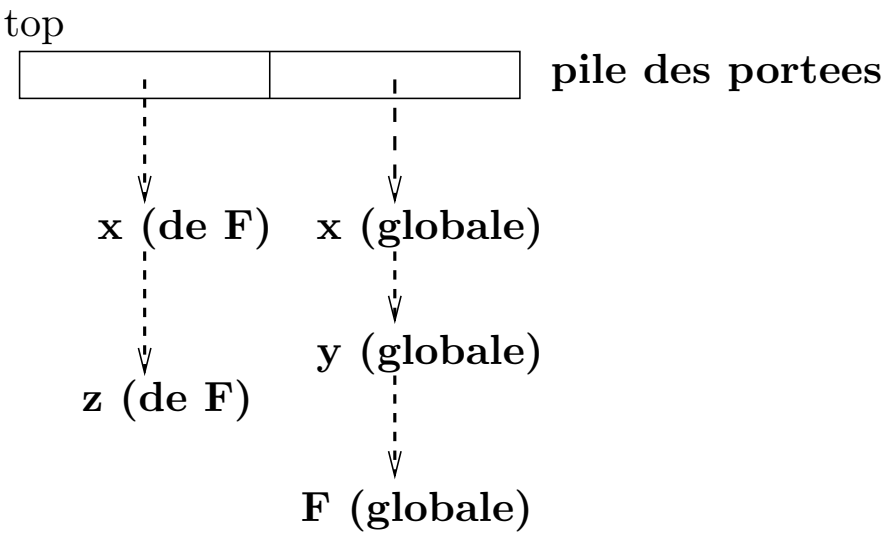
- en un point du programme, on a généralement plusieurs portées actives (bloc courant, blocs englobants, éventuellement membres de la classe ou entités globales)
- les portées ne se chevauchent pas, mais s'englobent : on peut donc les gérer en pile
  - seule exception, l'héritage : la portée d'un membre de classe s'étend aux classes filles
- la recherche d'un identificateur se fait donc de la portée en sommet de pile jusqu'à celle en fond de pile
  - donc si un même identificateur est apparaît dans plusieurs portées, on trouvera le plus englobé d'abord, ce qui correspond aux règles de visibilité

Exemple d'emboîtement de portées

- soit le programme C

```
int x; double y;
int F (int z) {
  int x; ...
```

- lors de l'analyse du corps de F, on aura (pour des portées implémentées en listes)



Exemple d'implémentation des portées

- toutes les portées ont une portée englobante, mais il y a celles qui ont une portée parente (les classes par exemple) et celles qui n'en ont pas (les blocs notamment)
- d'où une implémentation possible sans utilisation explicite d'une pile

```
public class Portee {
  protected Hashtable <String, Entite> table;

  public Portee englobante; // 'public' faute de place pour accesseur

  abstract public void entrer (String id, Entite en);
  abstract public Entite chercher (String id);
}
public class PorteeBloc extends Portee {
  public void entrer (String id, Entite en) // entre (id,en) dans table
  public Entite chercher (String id) // cherche id dans table
}
public class PorteeClasse extends Portee {
  protected PorteeClasse mere;
  public void entrer (String id, Entite en)  // entre (id,en) dans table
  public Entite chercher (String id) // cherche id dans table puis mere
}
```

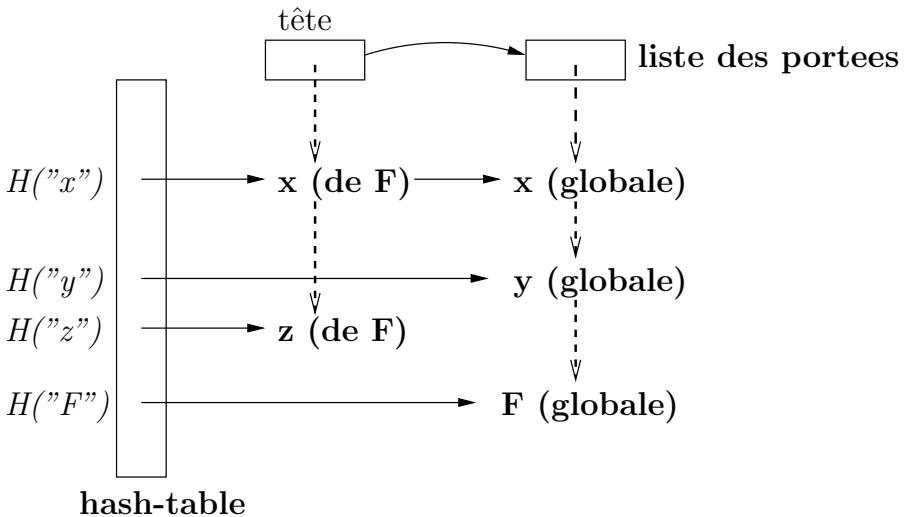
La table des symboles

- en grammaire attribuée, la table des symboles serait un attribut synthétisé (analyse des déclarations) et hérité (analyse des utilisations)
- si on programme la sémantique, ce peut être une variable globale (C), un attribut statique (Java), ou un paramètre des fonctions d'analyse
- une implémentation possible

```
public class TableSymbole {
  protected Portee courante; // le "sommet" de pile

  public void EntrerPortee (Portee p) {
    p.englobante = courante; courante = p;
  }
  public void QuitterPortee () { courante = courante.englobante; }

  public void entrer (String id, Entite en) { courant.entrer (id, en); }

  public Entite chercher (String id) {
    Entite en = null;
    for (Portee p = courante; p != null && en == null; p = p.englobante)
      en = p.chercher (id);
    return en;
  }
}
```

Table des symboles avec accès direct

- on peut éviter d'avoir à parcourir les différentes portées pour la recherche en définissant une table à double entrées
  - toutes les entités d'une même portée sont dans une liste simple (même pas ordonnée)
  - une table à adressage dispersé permet d'accéder directement à l'entité visible




```
int x ; double y ;
int F (int z) {
 int x ; ...
```

- noter que si l'analyseur lexical peut donner un n^o unique à chaque identificateur, $H(id)$ est ce n^o, et il n'y a pas de collisions

Comparaison des implémentations de table de symboles

- Pour m portées comprenant en moyenne n identificateurs

	Table directe	Pile d'arbres	Pile de listes
Entrée de portée	$O(1)$ ou $O(n)$	$O(1)$	$O(1)$
Sortie de portée	$O(n)$	$O(1)$	$O(1)$
Déclaration d'ident.	$O(1)$	$O(log_2 n)$	$O(n)$
Recherche d'ident.	$O(1)$	$O(m \times log_2 n)$	$O(m \times n)$

la déclaration d'ident implique que cet ident n'est pas déjà déclaré dans cette portée, donc une recherche dans la portée

l'entrée et la sortie d'une portée équivalent à empiler/dépiler cette portée (en fait un pointeur sur la portée)

pour la table à accès direct, l'entrée est $O(n)$ si les identificateurs ont été entrés dans la portée dans une passe précédente (il faut alors les mettre en front des listes de déclaration d'un même ident), sinon $O(1)$

Analyse de noms

- dans certains langages (ceux qui ne permettent pas les références en avant), il est possible de faire l'analyse d'utilisation de noms (qui consulte la table des symboles) en même temps que l'analyse des déclarations (qui remplit la table des symboles)
  - de fait, cela peut se faire parallèlement à l'analyse syntaxique
- si les utilisations en avant sont permises dans une portée, il est nécessaire d'effectuer les déclarations avant de vérifier les utilisations
  - dans ce cas, il faut faire une première passe sur les déclarations
- dans un langage comme Java
  - on fera une première passe pour entrer les classes comme types dans la portée globale
  - pour chaque classe, une seconde passe entrera ses attributs et ses méthodes dans sa portée
  - enfin, pour chaque méthode, on pourra faire l'analyse d'utilisation, parallèlement à l'entrée des variables locales dans leur portée

Analyse des déclarations globales

- déclaration d'un identificateur : 2 cas
  - s'il n'y a pas de surcharge, entrer un couple *(ident, entité)* dans la table des symboles provoque une erreur si l'identificateur existe déjà dans la portée courante
  - sinon, il faut vérifier que les types diffèrent : par exemple pour des méthodes Java, qu'elles aient un nombre différent de paramètres ou au moins un paramètre de types différents
  - éventuellement, vérifier si une variable de même nom n'est pas déclarée dans une portée englobante (Java)
- la table des symboles sera une variable globale (C+Bison) ou un attribut (statique) de la classe de l'analyseur (JavacCC+Java)
  - cette table est initialement constituée de la seule portée globale (vide au début)
  - de même pour les descripteurs des types primitifs (variable ou attribut `typeEntier, typeReel ...`) qui doivent être créés avant le début de l'analyse

Analyse des déclarations globales (suite)

- parallèlement à l'analyse syntaxique, au fur et à mesure de l'analyse des déclarations de classes (ou d'autres objets globaux), on entre ces déclarations dans la table (donc dans la portée globale)
- les autres déclarations (attributs, paramètres, variables locales) construisent un sous-arbre *decl*(*type*, *ident*, *expression*)
- pensez à

```
class A {
  void M () {
    int i = a + 1;
    ...
  }
  A() {}
  A (int x) { a = x; }
  int a = 10;
  ...
```

Arbre des déclarations locales

- dans un sous-arbre *decl*(*type*, *ident*, *expression*), le type est soit le descripteur d'un type primitif (reconnu par mot-clé), soit un ident

```
typedef struct {
  bool typePrimitif;
  union {
   Entite* type; /* si type primitif (int, float ...)*/
   const char* typid; /* si le type est un ident (classe) */
  }
} TypeOuIdent;
```

- et le sous-arbre est un nœud de type

```
typedef struct {
  TypeOuIdent type;
  const char* ident;
  Expression* init; /* null si pas d'expr. d'initialisation */
} Decl;
```

Construction d'une déclaration locale de variables

un arbre de déclarations peut être construit comme suit (en Bison)

```
%union {
  ...
  TypeOuIdent  toi;
  struct { ListDecl* d; TypeOuIdent t; } dec;
}
...
%type<toi> type
%type<dec> decl
%%
...
decl : type IDENT                { $$.d = newListDecl(newDecl($1, $2, NULL));
                                   $$.t = $1; }
     | type IDENT '=' expr       { $$.d = newListDecl(newDecl($1, $2, $4));
                                   $$.t = $1; }
     | decl ',' IDENT            { $$.d = append($1.d, newDecl($1.t, $3, NULL);
                                   $$.t = $1.t; }
     | decl ',' IDENT '=' expr { $$.d = append($1.d, newDecl($1.t,$3,$5);
                                   $$.t = $1.t; }
     ;
type : INT   { $$.typePrimitif = true; $$.type = &typeEntier; }
     | ...
     | IDENT { $$.typePrimitif = false; $$.typid = $1; }
     ;
```

Entrée et sortie de portée

l'analyse de noms (déclarations et utilisation) se faisant sur l'arbre abstrait, il faut décider quand on ouvre et quand on ferme une portée

- dans la passe d'analyse des déclarations des membres de classes
  - on boucle sur chaque classe : on entre dans la portée de chaque classe avant d'analyser les déclarations de ses membres, et on en sort à la fin
  - de manière très synthétique

```
pour toute classe C dans la liste des classes
 table.entrerPortee (C.portee);
 pour toute déclaration de membre M de C (attr. ou méth.)
  résoudre les noms de type (c'est à dire si le type est
  l'ident A, donner à M le type de la classe nommée A)
  entrer M dans la table (donc dans la portée de la classe)
 table.quiterPortee();
```

- dans la passe d'analyse des corps de fonction/méthodes et des blocs internes, on entre dans la portée au début de l'analyse de l'arbre, et on en sort à la fin

Analyse d'utilisation (corps de méthodes)

- la table des symboles contient initialement la portée globale, et c'est
  - un attribut hérité (si on utilise un système d'évaluation d'attributs)
  - une variable globale (ou un attribut statique) si on effectue l'analyse « à la main », ou éventuellement un paramètre des fonctions d'analyse

  et elle sera gérée *dernier entré premier sorti*

```
pour chaque classe C
 table.entrerPortee(C.portee) // on "empile" attributs
                              // et méthodes de la classe
 pour chaque méthode M de C
  table.entrerPortee(M) // la portée de M contient
                        // déjà ses paramètres
  pour chaque instruction I du corps de M analyser(I)
  table.quiterPortee() // on sort de la méthode

 table.quitterPortee() // on sort de la classe
```

Analyse des instructions d'un bloc

- la fonction d'analyse de nom des instructions est assez simple
  - si l'instruction est une déclaration *(type ID expr)* (voir page 21 pour *type*), alors

```
analyser(expr)
si le type est un ident
 soit T le type dénoté par cet ident
      (erreur si ce n'est pas un ident de type)
sinon type est le type T (un type primitif)
soit V une nouvelle entité variable de type T
table.entrerDecl(ID, V) // entre la variable en
                        // "sommet de pile" des portées
```

  - si l'instruction est un bloc B

```
table.entrerPortee(B.portee)
pour chaque instruction I de B analyser(I)
table.quitterPortee()
```

  - autres instructions : analyser leurs composants (sous-arbres), par exemple pour un nœud *if (expr instr)*, analyser *expr* et *instr*

Analyse d'utilisation des idents

- un identificateur `i` peut dénoter une entité de deux manières
  - il n'est pas qualifié (`i` tout simplement) : il suffit de chercher dans la table des symboles `e = table.chercher(i)`
  - il est qualifié (par exemple `a.m(...).b.i`) : il faudra d'abord analyser son préfixe `a.m(...).b`
    - `b` doit être une entité qui possède une portée (en Java, ce peut être une classe ou un paquetage : `java.lang.Math.PI`)
    - `i` doit être déclaré dans la portée de `b`
  - en fait tout peut être représenté par le même type de nœud

```
class Denotation {
  // arbre construit par l'analyse syntaxique
  Denotation prefixe; // nul si pas de préfixe
  String id;
  List<Expression> args; // nul si ce n'est pas un appel

  // arbre décoré par l'analyse sémantique
  Entite ent; // sera initialisée pas l'analyse de nom
  Type typ;   // sera initialisée pas l'analyse de type
```

Analyse d'utilisation des idents

(suite)

- l'analyse d'un nœud `Denotation` est donc

```
si this.prefixe == null
  this.ent = table.chercher(this.id)
sinon
  soit P le prefixe, analyser P
  si P.ent != null
    si P.ent possède une portée
      this.ent = P.portee.chercher(this.id)
    sinon erreur
```

- quelles entités possèdent une portée ?
  - les variables/attributs/paramètres de type classe
  - les méthodes dont le résultat est de type classe
  - et les classes internes, les paquetages ...

Traitement des importations

- dans la plupart des langages, les entités déclarées dans des modules (paquetages Java, espaces de nom C++ ...) doivent être dénotées en les préfixant par le nom du module
  - ceci permet qu'un même identificateur soit déclaré dans des modules différents soit qu'il y ait confusion pour une programme utilisant ces modules
- comme cela donne une écriture fastidieuse, il est possible d'« injecter » les identificateurs d'un module dans la portée globale (ne pas confondre avec le `#include` de C qui intervient lors d'un pré-traitement et correspond à un *copier-coller*)
- chaque module a une table de ses symboles : il suffit
  - d'entrer les classes (munies de leur table de symboles) dans la portée globale (Java)
  - d'entrer les entités présentes dans cette table au niveau global (Ada, C++ ...), en détectant les collisions de noms entre entités provenant de modules différents

Traitement des contrôles d'accès

- en cas de marque de contrôle d'accès (public, privé . . .), celle-ci est un champ des descripteurs d'entités, de même que la classe à laquelle elles appartiennent
- connaissant la classe courante (éventuellement son paquetage) et ses relations avec les autres classes, il est facile de vérifier le respect des règles d'accès, par exemple
  - entité de la classe courante : OK pour tous les accès
  - entité d'une autre classe
    si c'est une classe ancêtre, erreur si accès privé
    sinon erreur si accès non public
- l'idée peut s'étendre à *static* (par exemple représenté par un champ booléen du descripteur d'entité)
  - si la méthode est *static* et que l'entité appartient à l'objet courant, alors elle doit être *static*
  - on sait que l'entité appartient à l'objet courant si c'est un membre de la classe courante (ou d'une ancêtre) dénoté sans préfixe, ou alors avec `this` pour préfixe

Surcharge et redéfinition de méthodes

- en cas de surcharge, il y a correspondance entre un identificateur et un ensemble d'entités (en général des méthodes)
- lorsqu'une nouvelle entité est entrée dans une portée, s'il existe déjà une entité de même nom, il suffit de vérifier que le contexte d'utilisation du nom permettra de choisir la bonne entité (sinon, reporter une erreur : redéclaration interdite)
  - on peut par exemple autoriser l'homonymie pour un attribut et une méthode, pourvu qu'une méthode soit toujours utilisée suivie de parantèses
  - et évidemment, l'homonymie des méthodes (et plus généralement des fonctions ou des opérateurs, voir Ada ou C++ par exemple), pour peu qu'à l'appel, on puisse choisir $\Rightarrow$
    - le nombre de paramètres diffère, ou
    - au moins un paramètre a un type différent de celui à la même position dans l'autre méthode, ou,
    - les résultats sont de types différents (Ada)

Contrôle de type sur l'arbre abstrait

- on suppose que l'analyse de nom a été effectuée, c'est à dire que l'arbre abstrait a été décoré par les entités dénotées
  - en fait, le plus souvent l'analyse de type peut être effectuée en parallèle avec l'analyse d'utilisation de noms, puisque dès qu'on connaît l'entité dénotée, on connaît son type (ou ses types en cas de surcharge)
- sauf cas particuliers (traitement de la surcharge en Ada, besoin d'effectuer une inférence de type), la vérification du bon emploi des types n'utilise qu'une grammaire S-attribuée
- par exemple pour le `if`, de nœud *if (expression instruction)*

```
analyser l'expression
analyser l'instruction
  reporter une erreur
```

notez que dans les langages de la famille C serait accepté tout type pour lequel il est défini un 0

Expressions

- les nœud *expression* partagent tous un champ `Type typ`
- par exemple pour un opérateur binaire de nœud *binaire (expr oper expr)*, sans surcharge des opérateurs par l'utilisateur

```
analyser l'expression de gauche, soit Tg son type
analyser l'expression de droite, soit Td son type
selon operateur
  cas plus, moins, mult, div :
    si Tg et Td entier ou reel alors
      si Tg et TD entier alors typ = entier sinon typ = reel
    sinon erreur
  cas ega, non egal :
    Tg et Td doivent être des types comparables
    typ = booleen
  cas sup supeg, inf, infeg :
    Tg et Td doivent être des types avec une relation d'ordre
    typ = booleen
  ...
```

- on pourrait aussi imaginer l'existence d'une table des opérateurs, qui associe un triplet *(type opérande gauche, type opérande droit, type résultat)* à chaque opérateur, et qui règle par conséquent la question des surcharges définies par l'utilisateur

Variables

- les variables étant souvent des entités dénotées, leur type est évidemment celui de leur déclaration
- mais il existe des variables anonymes, par exemple
  - les variables indicées, de nœud *indexation (denotation expression)* :
    la dénotation doit évidemment être de type tableau (et l'expression de type compatible avec les entiers, en général)
    le type de la variable est le type des éléments du tableau
  - les variables dénotées par un pointeur (`*p` en C par exemple), de nœud *dereferenciation (denotation)* : la dénotation doit évidemment être de type pointeur, et le type de la variable est le type des éléments pointés
- *lvalue* et *rvalue* : la définition d'une variable peut contenir un modificateur de type indiquant qu'elle ne peut pas être modifiée (*const* en C/C++, *final* en Java)
  ce modificateur doit faire partie du type synthétisé pour effectuer les vérifications nécessaires (pour une affectation notamment)

Appel de fonction/méthode

sans surcharge

- c'est une dénotation avec une liste d'expressions (les arguments d'appel)
- l'analyse de nom a permis de déterminer l'entité, et donc son type
  - si l'entité n'est pas une fonction/méthode, reporter une erreur évidemment
  - sinon, parcourir en parallèle la liste de paramètres formels (qu'on trouve dans le prototype de la fonction) et la liste d'arguments
    - si le type d'un argument n'est pas compatible avec celui du paramètre formel, reporter une erreur
    - si les 2 listes n'ont pas la même longueur, reporter une erreur (sauf si le langage autorise des arguments par défaut, par exemple C++)
  - le type de l'expression est le type du résultat de la fonction

Appel de fonction/méthode

avec résolution de surcharge

- l'identificateur dénote un ensemble d'entités, et chacune a un type différent des autres
- en ne retenant que les fonctions/méthodes, opérer comme précédemment pour chacune d'entre elles successivement :
  si une des méthodes aboutit a un succès, la retenir
- sinon, il faut regarder les conversions possibles, et recommencer l'analyse parallèle des arguments et des paramètres formels en tenant compte des conversions possibles et en les comptant (selon les langages, il peut exister plusieurs types de conversions : chaque type de conversion a un poids, et on cumule ces poids)
  s'il existe une (et une seule) méthode ayant un coût de conversion inférieur aux autres, la retenir, sinon reporter une erreur