Compilation

Analyse sémantique statique

Tables des symboles

SI4 — 2018-2019

Erick Gallesio

Introduction

Analyse sémantique statique

- Elle concerne tous les aspects qui ne sont pas décrits dans la grammaire du langage.
- Elle vérifie que le programme lu est correct vis-à-vis de la définition du langage.

Le travail réalisé consiste principalement

- à vérifier qu'un identificateur est bien déclaré (*analyse de nom*)
- à vérifier que dans une expression, les opérandes d'un opérateur sont compatibles avec ce dernier (*analyse de type*)
- à trouver l'opérateur ou la fonction à appeler pour les langages avec surcharge,
- …

Analyse sémantique

Les langages de programmation contiennent des **entités** de différentes sortes:

- types,
- variables,
- fonctions,
- classes.

On représentera chacune de ces entités

- par une classe si le compilateur dispose d'objets,
- par une structure sinon (comme vu dans cours précédent).

Analyse de noms:

Notions de *portée* et de *visibilité* d'un identificateur.

La représentation interne des identificateurs d'un programme passera par une *table des symboles*.

Analyse de types:

Notions d'équivalence de type, de compatibilité, de surcharge, de conversion.

Représentation des entités

Ce qu'il faut pour représenter les différentes entités

Les types:

- **types simples** (entiers, booléens, …):
  - *ils sont prédéfinis dans le compilateur*
  - *les règles de conversions éventuelles sont câblées dans le compilateur (e.g. char → entier en C, entier → réels dans une expression).*
- **types composés** (structures, unions, fonctions, ….)
  - *ils sont basés sur d'autres types et, in fine, sur des types simples.*
  - *on a une représentation interne différente pour chaque type composé.*
  - *Représentation interne:*
    - tableaux: taille et type des éléments (éventuellement bornes et type des indices pour certains langages)
    - classes: lien d'héritage(s), membres, méthodes
    - …

Les autres entités d'un langage ont toujours un type:

- **variables**: le type suffit pour l'analyse statique
- **fonctions/méthodes**: prototype (type de retour et paramètres) et l'arbre abstrait de leur corps.

Exemples en toy

```
struct ast_node {
  enum node_kind kind;                    ///< kind of node
  ast_node *type;                         ///< node type (stmt: NULL)
  void (*produce_code)(ast_node *this); ///< method for code
  void (*analysis)(ast_node *this);     ///< method for analysis
  ...
};


struct s_type {
  ast_node header;    ///< AST header
  char *name;         ///< external name of the type ("int", ...)
  char *default_init; ///< Default init value  ("0", "NULL", ...)
  bool is_standard;   ///< type is predefined  (int, bool, ...)
};


struct s_class {
  ast_node header;     ///< AST header
  ast_node *name;      ///< Class name
  ast_node *parent;    ///< Class parent
  List members;        ///< Members of the class
  List vtable;         ///< Class virtual methods list
};
```

Portées d'un identificateur (1/3)

La plupart des langages de programmation permettent de définir des variables qui ne sont utilisables que dans une partie du programme seulement (**notion de portée**).

En général:

- Les **paramètres** d'une fonction, les **variables locales** ou les **variables d'un bloc** :
    - *ne sont utilisables que pendant l'exécution de la fonction/bloc;*
    - *peuvent masquer des variables de même nom définies à l'extérieur de la fonction/bloc (**notion de visibilité**)*
- Par ailleurs, certains langages objets permettent d'imposer des restrictions sur les accès à un identificateur: public, privé, … (**notion de contrôle d'accès**)

Portées d'un identificateur (2/3)

Quelques exemples de portée de l'identificateur `x`

```
class A {...int x;...} // portée de x = corps de A + descendance
class x {...}          // portée de x = fichier
class C {... x t; ...} // la classe x précédente


class B1 extends A {   // ici A::x masque la classe x globale
  x t ;                // erreur: x n'est pas un type dans A
 ...
}


class B2 extends A { // ici A::x masque la classe x globale

  void M () {
     A v;
     bool x;          // ce x masque A::x dans M
     ... x ...        // le x visible (celui de M)
     ... v.x ...      // le x de A
  }
}
```

Classes, fonctions/méthodes et blocs définissent chacun une portée:

- on "ouvre" une nouvelle portée à l'entrée, et
- on "ferme" cette portée à la sortie.

Portées d'un identificateur (3/3)

Une portée est une **association** entre:

- un identificateur et
- l'entité qu'il dénote

Cette association est établie pour une partie du programme à analyser.

Il faut implémenter les portées pour les constructions qui permettent de déclarer des identificateurs de portée locale à la construction:

- structures/unions (champs),
- classes (membres),
- blocs ou boucles (variables),
- unités de compilation (portée globale),
- modules …

Emboîtements des portées

- En un point du programme, on a généralement plusieurs portées actives:
  - *bloc courant*
  - *bloc englobant*
  - *membres de la classe*
  - *entités globales*
- les portées s'emboîtent les unes dans les autres (sauf en cas d'héritage où la porté d'un membre s'étend aux classes dérivées).

```
{ int a;
  ...
  for (int i = 0; i < 10; i++){ bool a; ... }
  for (int a = 0; a < 12; a+++{ bool i; ... }
  ...
}
```

On peut donc gérer les portée **en pile** (l'héritage sera traité à part)

La recherche d'un identificateur se fait donc

- de la portée en sommet de pile (bloc le plus interne),
- vers la portée en fond de pile (variables globales)

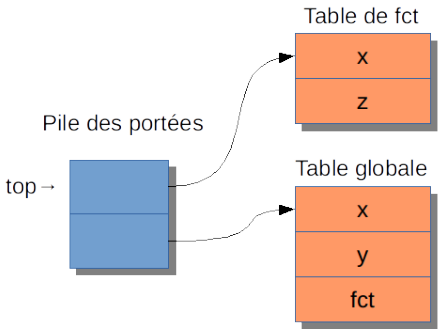
Emboîtements des portées (exemple)

Soit le programme suivant:

```
int x, y;

int fct(int z){
    int x = 2 * y;
    return x * z;
}
```

Lors de l'analyse de la fonction `fct`, on aura:



Implémentation des portées *Toy* (1/2)

La table des symboles est représentée par une liste chaînée de *hash tables* génériques (qui associent chaîne → information)

```
struct symbol_table {
  Hash_table table;          // A hash table
  struct symbol_table *next; // Embedding block (or parent class)
};

static Symbol_table current_table  = NULL;

/// Create a new symbol table for current scope.
void enter_scope(void);

/// Mark the table for current scope for destruction.
void leave_scope(void);

/// Declare an object (ident or function) in the current block.
void symbol_table_declare_object(char *id, ast_node *obj);

/// Search a symbol starting from current scope.
ast_node *symbol_table_search(char *id);
```

- **enter_scope** empile une nouvelle *hash table*;
- **leave_scope** la dépile.

Implémentation des portées *Toy* (2/2)

```
//
// Create a new symbol table for current scope.
//
void enter_scope(void){
  Symbol_table tmp = must_malloc(sizeof(struct symbol_table));

  tmp->table    = hash_table_create();
  tmp->next     = current_table;
  current_table = tmp;
}


//
// Search a symbol starting from current scope.
//
ast_node *symbol_table_search(char *id) {
  Symbol_table t;

  for (t = current_table; t; t = t->next) {
    ast_node *res= hash_table_search(t->table, id);
    if (res) return res;
  }
  return NULL;
}
```

Analyse des déclarations

Déclaration d'une variable:

- dans un langage "classique", c'est simple:
  - *on cherche l'identificateur dans la portée courante:*
    - s'il existe déjà c'est une erreur;
    - sinon, on le rentre dans la table de la portée.

```
// Declare an object (ident or function) in the current block.
void symbol_table_declare_object(char *id, ast_node *obj) {
  ast_node *old = hash_table_search(current_table->table, id);

  if (old)
    error_msg(obj, "%s is already declared", id);
  else
    hash_table_add(current_table->table, id, obj);
}
```

- dans une langage avec surcharge
  - *vérifier qu'il n'existe pas un identificateur identique dans la portée (e.g. pour une méthode: types de paramètres ≠ ou nombre de paramètres ≠).*

Analyse d'utilisation des identificateurs

Un identificateur `id` peut dénoter une entité de deux manières:

- si il est **non qualifié**: il suffit simplement de le chercher dans la table des symboles.
- si il est **qualifié**: (e.g. `a.foo(...).x.id`) il faut d'abord analyser son préfixe `a.foo(...).x`:
  - *Vérifier que `x` possède une portée (par exemple que c'est une instance de classe);*
  - *Vérifier que `id` est bien déclaré dans cette portée.*

Les entités qui possèdent une portée:

- structures, unions
- variables/attributs paramètres de type classe;
- méthodes dont le résultat est de type classe;
- modules, …

Analyses spéciales (1/2)

Importations

Dans la plupart des langages,

- les entités déclarées dans un module doivent être dénotées en les préfixant par le nom du module
- possibilité *d'injecter* les identificateurs du module dans la portée globale

En Python, par exemple

```
import foo
from bar import *
from foobar import X
from gee import Y as Z
```

Implémentation:

- chaque module a sa table des symboles;
- on entre les objets globaux du module dans la porté globale;
- contrôler qu'il n'y a pas de collision, éventuellement renommer.

Analyses spéciales (2/3)

Contrôles d'accès

Si le langage propose du contrôle d'accès (`public`, `private`, ...) sur les membres des classes.

- utiliser un champ des descripteurs d'entité pour noter le contrôle d'accès.
- connaissant la classe courante et ses relations avec les autres classes, on peut vérifier le respect des règles d'accès.

Par exemple:

- *entité de classe courante: elle peut accéder à tout*
- *entité d'une autre classe*
  - si classe dérivée: erreur si accès `private`
  - si classe quelconque: erreur si l'accès n'est pas `public`

Analyses spéciales (3/3)

Surcharge

La surcharge permet à un identificateur de désigner un ensemble d'entités.

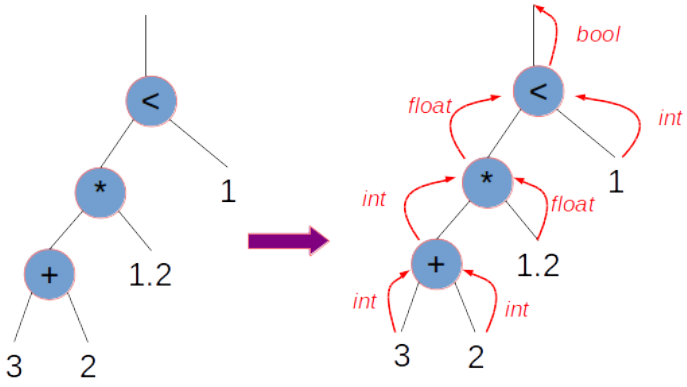
Lorsqu'on on entre une nouvelle entité dans la portée:

- vérifier que le contexte d'utilisation permettra le choix de l'entité parmi tous ses "homonymes".
  - *par exemple sur une méthode, vérifier que le nombre de paramètres ou que le type de ses arguments diffère, voire son résultat.*
  - *Implementation possible avec du* ***name mangling****.*
- si le choix ne peut être fait statiquement (i.e. par le compilateur), signaler une erreur indiquant que la redéclaration du symbole est interdite.

Compatibilité de types

Le contrôle de types:

- se fait en parcourant l'arbre abstrait
- peut se faire généralement en même temps que l'analyse de noms



Compatibilité de types en Toy

La fonction qui vérifie la compatibilité de types de deux expressions en *Toy-base*:

```
static bool compatible_types(ast_node *e1, ast_node *e2) {
  ast_node *t1 = AST_TYPE(e1), *t2 = AST_TYPE(e2);

  if (!t1 || !t2) return true; // to avoid error cascades => true
  if (t1 == t2)   return true; // same types => true

  // Testing the mix of integer and float computations
  if ((t1 == int_type   && t2 == float_type) ||
      (t1 == float_type && t2 == int_type))
    return true;

  return false;
}
```

Le test sur NULL permet d'éviter les cascades d'erreurs:

- si on ne sait pas typer un composant (variable non déclarée par exemple), on fixe son type à NULL.

Contrôle appel de fonction/méthode (1/2)

Si le langage n'a **pas de surcharge**, c'est assez simple:

- parcours la liste de paramètres formels et la liste de paramètres effectifs en parallèle et vérifier la compatibilité 2 à 2.

```
void parameter_cmp(List formal, List effective, ast_node *node) {
  List_item  p1 = list_head(formal);
  List_item  p2 = list_head(effective);

  for (int idx=1;
       p1 && p2;
       p1 = list_item_next(p1), p2 = list_item_next(p2), idx++){
    if (!compatible_types(list_item_data(p1), list_item_data(p2))){
      error_msg(node, "type of parameter #%d is not compatible"
                      " with previous definition", idx);
    }
  }

  if (p1 && !p2) error_msg(node, "call has not enough parameters");

  if (!p1 && p2) error_msg(node, "call has too much parameters");
}
```

- vérifier que le type du résultat est correct dans le contexte d'appel

Contrôle appel de fonction/méthode (2/2)

Si le langage **permet la surcharge**:

- L'identificateur dénote un ensemble d'entités de types différents.
- Appliquer la fonction de compatibilité sur tous les candidats possibles.
  - *si il n'y a qu'une méthode/fonction compatible, la retenir*
  - *Si plusieurs fonctions/méthodes restent, recommencer l'analyse des paramètres formels/effectifs en tenant compte des conversions possibles en les comptant. - si il existe une seule méthode/fonction ayant un coût de conversion inférieur aux autres, la retenir - sinon, signaler une erreur.*
- produire le code d'appel de la méthode retenue.
