Projet de compilation (Étape 2)

Samuel Tardieu
SE202 - Année scolaire 2016/2017

Deuxième étape

Avant de commencer

Avant de commencer à travailler, il vous faut intégrer le code venant de l'équipe pédagogique. Depuis votre dépôt :

```
% git pull template step2
```

À ce stade, votre dépôt a été enrichi de l'étape `step2` du dépôt de référence. Vérifiez que les tests de base fonctionnent toujours :

```
% workon se202
(se202)% python -m unittest
```

La première commande fait appel à votre environnement de développement virtuel `tiger`, si ce n'était pas déjà fait. La seconde cherche tous les tests unitaires du dépôt et les exécute.

Travail à faire

Vous allez ajouter, avant la séance prochaine, les notions de

- variable ;
- fonction ;
- type (uniquement `int` pour l'instant) ;
- scope

dans un **nouveau** visiteur `Binder` dont la partie commune vous est fournie dans `semantics/binder.py`. Toute erreur dans ce visiteur devra lever une exception de type `BindException`.

Vous rajouterez ensuite dans la grammaire :

- les commentaires ;
- le moins unaire.

Évaluation de l'arbre

Contrairement à l'étape précédente, il ne sera pas demandé d'enrichir le visiteur qui évalue l'arbre. En effet, le but n'est pas de faire un interpréteur Tiger mais un compilateur, donc nous ne ferons pas d'interprétation complète.

Variables et fonctions

Les variables et fonctions sont déclarées dans le corps d'un bloc `let`, avec les mots clés `var` et `function`, comme indiqué dans les transparents de cours.

Exemple :

```
let
  var a: int := 3
  var b := a + 1
  function add(x: int, y: int): int = x + y
in
  add(a + b, 3)
end
```

Les types sont facultatifs sauf pour la déclaration des paramètres d'une fonction.

Notes

- Il y a au moins une déclaration de variable ou de fonction entre `let` et `in`.
- On n'acceptera, pour l'instant, qu'une et une seule expression entre `in` et `end`, mais on la mettra dans une liste Python en prévision de la suite.
- Lorsqu'un type n'est pas donné explicitement, on le stockera dans l'arbre comme étant `None`. S'il est donné explicitement, on utilisera un nœud `Type` pour l'attribut correspondant.
- Lors de la déclaration d'une fonction, les arguments seront du type `VarDecl` (comme une déclaration de variable) avec une expression positionnée à `None`.

Détection de listes (1 ou plus)

Comment faire dans la grammaire pour détecter des listes de 1 ou plusieurs déclarations (supposons que la règle pour une déclaration s'appelle `decl` et recouvre les déclarations de variable et de fonction) ?

```
def p_decls(p):
    '''decls : decl
             | decls decl'''
    p[0] = [p[1]] if len(p) == 2 else p[1] + [p[2]]
```

En testant la taille de la liste `p`, on peut distinguer si on est dans la première alternative de la règle (déclaration simple, `len(p) == 2`, et donc on construit une liste ne contenant que cette déclaration) ou dans la seconde (on construit une liste avec la liste correspondant à `decls` à laquelle on ajoute la nouvelle déclaration).

Détection de listes (0 ou plus)

Dans le cas des listes acceptant un nombre quelconque de termes (0 ou plus) séparés par un symbole (par exemple une virgule), la règle est un peu plus complexe car il ne faut pas se retrouver avec le séparateur seul. Par exemple, pour les arguments lors de l'appel d'une fonction :

```
def p_args(p):
    '''args :
                | argssome'''
    p[0] = p[1] if len(p) == 2 else []


def p_argssome(p):
    '''argssome : expression
                    | argssome COMMA expression'''
    p[0] = [p[1]] if len(p) == 2 else p[1] + [p[3]]
```

Profondeur et échappement

En Tiger, il est possible d'accéder depuis une fonction à une variable définie en dehors de cette fonction :

```
let
  var a := 3
  function f() =
    let
      function g() = a // a défini en dehors de g et f
    in
      g()
    end
in
  f()
end
```

Déclaration et visibilité

La visibilité des nouveaux identifiants diffère selon qu'il s'agit d'une variable ou d'une fonction :

- Une fonction peut s'appeler elle-même depuis l'intérieur de son corps (récursivité).
- Une variable est disponible juste après sa déclaration. On ne peut pas avoir par exemple `let a := a in 0 end` car `a` ferait référence à elle-même. Bien entendu, s'il y a une variable de même nom dans un scope plus extérieur, celle-ci peut être référencée.

```
let var a := 10 in
  let var a := a + 1 in  // Référence au a précédent
    a * 2                // Vaut 22
  end
end
```

Profondeur et échappement

Lorsqu'elle s'exécute sur le microprocesseur, une fonction a en général accès à :

- ses arguments ;
- ses variables locales ;
- les variables globales du programme ;
- un pointeur vers le contexte de la fonction du niveau supérieur (lien statique, ou *static link*).

Dans l'exemple précédent, `g` devra pour accéder à `a` déréférencer deux fois son *static link*: une fois pour accéder au contexte de `f` (qui englobe `g`) puis une seconde fois pour accéder au contexte contenant `a`.

Profondeur et échappement

Pour faciliter la génération de code et calculer ces déréférencements, on procèdera lors de l'analyse sémantique à différentes opérations :

- un compteur d'imbrication de fonctions sera maintenu lors de l'analyse sémantique (incrémenté lors de l'entrée dans une déclaration de fonction, décrémenté ensuite) ;
- chaque déclaration enregistrera la valeur actuelle du compteur dans un attribut `depth` de sa structure (nœuds `VarDecl` ou `FunDecl`) ;
- chaque utilisation enregistrera la valeur actuelle du compteur dans un attribut `depth` de sa structure (nœuds `Identifier`).

Profondeur et échappement

Lors de la génération de code, le compilateur pourra librement choisir de stocker une variable en mémoire ou dans un registre (pour un accès plus rapide). Cependant, si la variable doit être accédée à travers le *static link*, elle doit nécessairement se trouver en mémoire, les registres étant utilisés par la fonction courante.

Pour cela, chaque déclaration possède un attribut `escapes` qui sera positionné à `True` lors du parcours de l'arbre si la variable est accédée depuis une profondeur supérieure à celle de sa déclaration. Dans ce cas, on demandera au générateur de code (plus tard dans le projet) de toujours placer la variable en mémoire plutôt que dans un registre.

Scopes

Le visiteur qui parcourt l'arbre pour effectuer l'analyse sémantique maintient à jour une liste courante de *scopes*, des dictionnaires qui contiennent le nom des variables définies et pointent vers leur définition.

Le fichier `semantics/binder.py` contient des méthodes permettant de commencer un nouveau *scope*, d'ajouter un nom dans le *scope* courant et de faire une recherche. De plus, ces fonctions se chargent de positionner les attributs `depth` et `escapes` des déclarations et des identifiants.

Appels de fonction

L'analyseur sémantique devra également vérifier, pour chaque appel de fonction (nœuds `FunCall`), que :

- l'identifiant utilisé est bien associé à une déclaration de fonction (on pourra utiliser la fonction standard `isinstance` de Python) ;
- le bon nombre d'arguments est passé à la fonction par rapport au nombre attendu (qu'on pourra trouver dans la déclaration).

Tests

Le visiteur d'affichage `parser.dumper.Dumper` s'est doté d'un nouveau paramètre dans son constructeur : un champ `semantics` (par défaut à `False`) qui, s'il est mis à `True`, doit afficher des informations complémentaires (uniquement pour les déclarations de variables et l'utilisation de ces variables, pas pour les déclarations ni les appels de fonction) :

- lorsque le champ `escapes` d'une déclaration est `True`, il faut afficher `/*e*/` juste après le nom de la nouvelle variable ;
- lorsque la profondeur d'un identifiant est supérieure à celle de sa déclaration, il faut afficher `/*n*/` après l'identifiant où *n* est le nombre (positif) représentant la différence de profondeur.

Tests

Par exemple, le code

```
let var a := 3
    function f() = let function g() = a in g() end
in f() end
```

s'affichera comme

```
let var a/*e*/ := 3
    function f() = let function g() = a/*2*/ in g() end
in f() end
```

Tests

Autre exemple :

```
let var a := 3
    var b := 4
    function f(c: int) = a + c
in f(b) end
```

donnera :

```
let var a/*e*/ := 3
    var b := 4
    function f(c: int) = a/*1*/ + c
in f(b) end
```

car `b` n'est pas utilisé à une profondeur de déclaration de fonction supérieure à sa déclaration.

Commentaires

On modifiera le parseur Tiger pour ignorer les commentaires :

- `//` commence un commentaire qui se termine à la fin de la ligne courante ;
- `/*` commence un comentaire qui se termine avec `*/`, ces commentaires pouvant être imbriqués.

Commentaires : implémentation

Pour implémenter les commentaires imbriqués, on pourra utiliser les états (*states*) de Ply dans le lexer :

- Un état `ccomment` gère le fait que nous soyons dans un commentaire délimité par `/*` et `*/`.
- On entre dans cet état et on le place (*push*) sur la pile des états depuis l'état normal ou l'état `ccomment` à travers la règle `t_ANY_begin_ccomment`.
- Dans cet état, on détecte, grâce à une règle spécifique, les caractères tant qu'on n'a pas `/*` ou `*/`.
- Dans cet état, `*/` enlève (*pop*) un état de la pile d'état pour revenir à l'état antérieur.

Du coup, si on a par exemple trois niveaux imbriqués de commentaire, trois états `ccomment` seront placés sur la pile des états et devront être enlevés avant de revenir à l'analyse classique.

Moins unaire

On ajoutera au parseur Tiger la gestion de l'opérateur - unaire :

- Il est plus associatif que la multiplication et la division.
- Il doit apparaître dans l'arbre comme une soustraction entre la constante 0 et son argument (on n'introduira pas de nouveau type de nœud).

La documentation de Ply indique comment lui affecter une priorité différente de celle du - binaire (chercher `UMINUS`).

```
 % ./tiger.py -d -e -E "-17+1"
 ((0 - 17) + 1)
Evaluating: -16
```