# Projet de compilation (Étape 3)

Samuel Tardieu
SE202 - Année scolaire 2016/2017

# Troisième 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 step3
```

À ce stade, votre dépôt a été enrichi de l'étape `step3` 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

- types (le code du *typer* est fourni par les enseignants) ;
- séquences d'expressions ;
- affectations ;
- `if`/`then` sans `else`
- les boucles `while` et `for` ;
- la sortie anticipée de boucles avec `break`.

# Évaluation de l'arbre

Comme à l'étape 2, 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.

Par contre, tout ce qui était supporté lors de la première étape par l'évaluateur doit continuer à fonctionner.

# Types

Un nouveau visiteur (`Typer`, dans `typer/typer.py`) rajoute des types explicites là où ils peuvent être omis. Un nouveau paramètre `-t` de `tiger.py` permet de les visualiser :

```
% ./tiger.py -dtE "let var a := 3 in a end"
let var a: int := 3 in a end
```

Les deux types que nous manipulerons seront `int` et `void` (qui représente l'absence de valeur). Un type `void` ne peut jamais être écrit explicitement, et n'est jamais affiché.

Vous devez modifier le visiteur `Dumper` pour ne pas afficher le type s'il est égal à `void`.

# Types

C’est la seule modification que vous aurez à faire concernant les types. Dans le reste de cet étape, les types sont indiqués afin de comprendre la sémantique de Tiger, mais vous n’aurez pas à les gérer vous-même.

Vous êtes invités à lire le code du typer fourni.

# Séquences

En Tiger, il est possible d'écrire une séquence d'expressions en les entourant d'une paire de parenthèses et en les séparant par des points-virgules. Une séquence évalue toutes ses expressions et renvoie sa dernière valeur. La séquence vide est autorisée (son résultat est alors de type `void`).

Modifier le parseur pour :

- supprimer le traitement actuel des parenthèses dans les expressions (il n'est plus nécessaire, une expression parenthésée étant une séquence avec une seule expression) ;
- reconnaître des séquences et construire un nœud de type `SeqExp` (voir `ast/nodes.py`).

# Affichage des séquences

Afin de rester compatible avec les étapes précédentes, le `Dumper` devra afficher les séquences ainsi :

- si une séquence n'a qu'une seule expression, on l'affiche sans parenthèses ;
- sinon, on affiche une parenthèse ouvrante, les expressions séparées par des points-virgules, puis une parenthèse fermante.

```
% ./tiger.py -dtE "(1)"
1          # 1 exp
% ./tiger.py -dtE "(1+2)"
(1 + 2)    # 1 exp ; les () viennent de BinaryOperator
% ./tiger.py -dtE "(1; 2)"
(1; 2)     # 2 exps
% ./tiger.py -dtE "()"
()         # 0 exp
```

# Vérification

À ce stade, vous devez toujours passer les tests des étapes 1 et 2 (ansi que quelques tests de l'étape 3). N'oubliez donc pas de modifier votre évaluateur.

En règle générale, vous devrez toujours pouvoir passer les anciens tests.

# Expressions multiples dans `let`

Dans une expression `let`, il est possible de mettre 0 ou plusieurs expressions entre le `in` et le `end`, séparées par des points-virgules :

```
let
  var a := 3
in
  a;
  a * 2
end
```

Les parenthèses ne sont pas nécessaires. Implémentez cela dans votre parseur, dumper, évaluateur (et binder si vous n'aviez considéré qu'une seule expression dans le champ `exps` d'un nœud `Let`).

# Affectations

Il est possible d'affecter une valeur à une variable (de type `var` ou un argument de la fonction courante) avec l'opérateur `:=` déjà utilisé pour une déclaration de variable.

```
let
  var a := 3
  function f(b: int) = (b := b + 1; a := a + b * 2)
in
  f(5)    // Maintenant a vaut 15
end
```

Une affectation est de type `void`.

Implémentez l'affectation dans le parseur, dumper et binder. Le nœud à créer est de type `Assignment`. Le binder devra vérifier que la variable affectée est bien de type `VarDecl`.

# Vérification de l'affichage

Normalement, le code de la page précédente doit s'afficher (à l'indentation et aux espace près) comme :

```
% ./tiger.py -dtE "let var a := 3 function f(b: int) =
(b := b + 1; a := a + b * 2) in f(5) end"
```

```
let var a/*e*/: int := 3
    function f(b: int) =
      (b := (b + 1); a/*1*/ := (a/*1*/ + (b * 2)))
in f(5) end
```

Notez qu'aucun type de retour n'est affiché pour `f` : la dernière expression était de type `void`, la fonction est de type `void`.

# `if/then` sans `else`

Il est possible d'omettre la partie `else` d'un test. Dans ce cas, la partie `then` doit être de type `void` et toute l'expression `if` est de type `void`.

Si aucune clause `else` n'est spécifiée, il faut utiliser `None` dans la partie correspondante (`else_part`) du nœud `IfThenElse`.

Modifier le parseur, dumper et binder. On ne vous demandera pas d'évaluer une expression sans `else` dans votre évaluateur.

# Boucles `while`

Tiger supporte les boucles `while` avec la syntaxe suivante :

```
while expression do expression
```

La première expression est le test à exécuter, la seconde (qui doit être de type `void`) ce qu'il faut exécuter. Le tout est de type `void`.

Modifier le parseur, dumper et binder. Le nœud à créer est de type `While`.

# Exemple de code avec `while`

```
let
  function fact(n: int) =
    let var result := 1 in
      while n > 0 do (result := result * n; n := n - 1);
      result
    end
in
  fact(5)
end
```

Vous remarquerez si vous affichez ce code en utilisant le paramètre `-t` de `tiger.py` que la fonction `fact` a été correctement identifiée comme retournant un `int`.

# Boucle `for`

La boucle `for` permet d'itérer sur un ensemble de valeurs. La syntaxe est :

```
for identifier := expression to expression do expression
```

Par exemple :

```
let function fact(n: int) =
  let var result := 1 in
    for i := 1 to n do result := result * n;
    result
  end
in
  fact(5)
end
```

# Boucle for

L'indice de boucle ne peut pas être modifié par une affectation. Pour le différencier d'une variable, nous utiliserons le type de nœuds `IndexDecl`.

Une boucle `for` crée un nouveau scope pour contenir la déclaration de l'indice de boucle. Par contre, les bornes doivent être évaluées dans le scope extérieur et ne peuvent bien évidemment pas faire référence à l'indice de boucle.

Modifier le parseur, dumper et binder. Le nœud à créer est de type `For`.

# Sortie anticipée avec `break`

Il est possible de sortir de manière anticipée d'une boucle `while` ou d'une boucle `for` en utilisant le mot-clé `break` (qui est une expression de type `void`).

# Sortie anticipée avec break : limitations

`break` est invalide dans la partie **déclarative** d'un `let` :

```
let var a := 0 in
  for i := 1 to 10 do
    let var b := 1 in
      if i > 3 then break;   // Valide (corps du let)
      a := a + b
    end;
  a   // Vaut 30
end

for i := 1 to 10 do
  let var a := (break; 1) in (a; ())   // Invalide
```

# Sortie anticipée avec `break`

Modifier le parser, dumper et binder. Le nœud à créer est de type `Break`.

Lors de sa traversée de l'arbre, le binder pourra utiliser les fonctions `push_new_loop()` et `pop_loop()` (voir le code et la documentation dans `semantics/binder.py`) pour se souvenir de la boucle la plus interne entourant le `break`. Le champ `loop` du nœud `Break` devra faire référence au nœud `For` ou `While` dont le `break` permet de sortir.

# C’est tout pour le moment

Lors de la prochaine étape, nous traduirons le code en *IR* (*intermediate representation*).