Pablo Rauzy <pr@up8.edu>
https://pablo.rauzy.name/teaching/ic

Interprétation et compilation

TP 4 : Un premier interpréteur

Dans ce TP :
— Écriture d'un premier interpréteur.

Exercice 0.

Mise en place.

1. Notre petit projet va consister en quatre modules :
   — `Ast` qui contiendra la définition de notre représentation intermédiaire ;
   — `Print` qui contiendra une fonction permettant d'afficher les valeurs ;
   — `Baselib` qui contiendra notre bibliothèque de base ;
   — `Interp` qui contiendra le code de l'interpréteur.
   Ces quatre modules seront utilisés par nos programmes que l'on définira chacun dans un autre fichier comme une valeur du type `Ast.prog` que l'on passera à la fonction `Interp.eval`.
2. Pour commencer, on ne va définir dans notre module `Ast` qu'un type `value` qui peut contenir des valeurs de nos types de base (`void`, `bool`, `int`, `str`).
   → Créez le fichier `ast.ml` et définissez y le type `value`.
3. → Créez le fichier `print.ml` et définissez y la fonction `print_value : Ast.value -> unit` qui affiche une valeur.
4. Pour l'instant notre langage ne peut que exprimer des valeurs constantes de nos types de base, donc l'évaluation est trivial.
   → Créez le fichier `interp.ml` et définissez y la fonction `eval_value : Ast.value -> Ast.value`.
5. → Dans un fichier de test (par exemple `test.ml`) écrivez un programme, et testez que tout fonctionne correctement en le compilant avec `ocamlbuild test.byte` puis en l'exécutant avec `./test.byte`.

Exercice 1.

Variables et environnement.

1. Notre objectif est maintenant de gérer les variables et les assignations. C'est très important de bien comprendre le mécanisme que l'on va mettre en œuvre ici, donc on va rester le plus simple possible sur tout le reste.
   → Commençons par définir un type `expr` dans notre module `Ast` qui peut être soit une valeur de type `value`, soit un nom de variable.
2. On veut maintenant que nos programmes consistent en une séquence d'instructions qui puissent être soit une assignation de la valeur d'une expression à une variable, soit une instruction de renvoie (`return`).
   → Définissez dans `ast.ml` le type `instr` et le type `block` qui est simplement une liste de `instr`.
3. Dans le module `Interp`, on veut maintenant définir trois nouvelles fonctions :
   — `eval_expr` qui évalue une expression ;
   — `eval_instr` qui évalue une instruction ;
   — `eval_block` qui évalue un bloc d'instructions.
   Commençons par la première. Il y a pour l'instant seulement deux cas dans notre langage : soit l'expression est une constante, soit c'est un nom de variable. Dans le cas où c'est une constante, il suffit d'appeler `eval_value` que l'ont a déjà écrite. Dans le cas où c'est un nom de variable, il faut aller chercher sa valeur quelque part…
   Lors de l'interprétation, nous allons avoir besoin d'un *environnement* pour y stocker les valeurs définies par le programme. Cet environnement contiendra initialement les valeurs prédéfinies par notre langage.
   Pour représenter un environnement, on a besoin d'un dictionnaire clef-valeur, où les clefs sont des chaînes de caractères (les noms des variables et fonctions).
   En OCaml, on peut utiliser le *foncteur* `Map.Make` de la bibliothèque standard pour créer un tel dictionnaire. Il suffit de lui passer le module `String` pour créer un module de dictionnaire (map) dont les clefs sont des chaînes de caractères. Voir la documentation sur `https://ocaml.org/api/Map.Make.html`.
   → Créez le fichier `baselib.ml` et définissez y le module `Env` :

```
1 module Env = Map.Make(String)
```

Une fois ce module défini on dispose notamment de :
— `'a Env.t` : le type “environnement” dont les valeurs sont de type `'a` ;
— `Env.empty` : un environnement vide ;
— `Env.add k v e` : renvoie un environnemt identique à `e` avec en plus l'association de `v` à `k` ;
— `Env.find k e` : renvoie la valeur associée à `k` dans l'environnement `e`.

4. → Définissez dans le module `Interp` la fonction `eval_expr : Ast.expr -> 'a Baselib.Env.t -> 'a` (sachant pour l'instant le type `'a` correspond forcément à notre type `Ast.value`).

5. On veut maintenant définir la fonction `eval_instr`. À nouveau il n'y a que deux cas à traiter. Soit l'instruction est un renvoie, et dans ce cas il suffit de renvoyer la valeur de l'expression, soit c'est une assignation, et dans ce cas il faut mettre à jour l'environnement.

   Dans un paradigme impératif, on pourrait imaginer un environnement global qu'on mettrait à jour de manière impérative. Il faudrait dans ce cas penser à rétablir l'état de l'environnement à chaque sortie de fonction pour en retirer les éventuelles variables locales par exemple.

   Dans le paradigme fonctionnel, notre environnement n'est pas mutable, il faut donc qu'on renvoie un nouvel environnement augmenté lorsqu'on évalue une assignation. Et ce sera à la fonction appelante (dans notre cas, `eval_block`) de prendre en compte le nouvel environnement pour la suite tant que c'est pertinent.

   On peut pour cela définir un type de résultat `'a res` qui permettra de définir si on renvoie une valeur de type `value` ou un environnement de type `'a Env.t` :

```
type 'a res =
  | Ret of value
  | Env of 'a Env.t
```

   (Le type est paramétrique car on aura plus tard plusieurs types de valeur possibles dans notre environnement).

   → Définissez dans le module `Interp` la fonction `eval_instr : Ast.instr -> 'a Env.t -> 'a res` (sachant pour l'instant le type `'a` correspond forcément à notre type `Ast.value`).

6. Il ne nous reste plus qu'à écrire la fonction `eval_block`. Un bloc est une liste, il y a donc deux cas à gérer : soit la liste est vide, et dans ce cas on doit simplement dire à l'exécution de poursuivre en renvoyant l'environnement courant, soit il reste au moins une instruction à traiter, et dans ce cas on utilise notre fonction `eval_instr` pour évaluer cette instruction et, selon le résultat, poursuivre ou non l'évaluation du bloc.

   → Définissez dans le module `Interp` la fonction `eval_block : Ast.instr list -> 'a Env.t -> 'a res` (sachant pour l'instant le type `'a` correspond forcément à notre type `Ast.value`).

7. On va maintenant définir temporairement notre fonction `eval` en faisant appel à `eval_block` avec un environnement de départ vide :

```
let eval prog =
  match eval_block prog Env.empty with
  | Ret v -> v
  | Env e -> Nil
```

   → Mettez à jour `test.ml` en utilisant la fonction `eval` pour tester votre travail.

Exercice 2.

Appel de fonction et bibliothèque de base.

1. Notre objectif est maintenant d'ajouter des fonctions natives à notre langage. Nous allons prendre pour exemple l'addition. Pour éviter les conflits de noms entre les *builtins* et les fonctions qui seront définies dans le code interprété, une astuce simple est d'utiliser en préfixe un caractère qui ne sera pas autorisé dans les identifiants de notre langage, par exemple `%`.

   On pourrait vouloir ajouter dans notre environnement directement une fonction OCaml native :

```
let baselib = Env.add "%add" (fun a b -> a + b) Env.empty
```

   Mais en faisant ça, on aura un environnement de type `(int -> int -> int) Env.t` et on ne pourrait plus mettre dans notre environnement que des fonctions qui prennent deux entiers et renvoie un entier.

   En pratique on veut évidemment pouvoir mettre dans notre environnement des fonctions d'arité et de types différents, mais également pouvoir continuer à y stocker nos variables.

   On va donc définir un type de fonctions natives qui prend un seul argument de type `Ast.value list` et qui renvoie une valeur de type `Ast.value` :

```
open Ast

type native = value list -> value
```

   Cela nous permet ensuite de définir un type `env_value` qui peut exprimer une valeur ou une fonction native :

```
type env_value =
  | V of value
  | N of native
```

Cela nous permet de mettre notre fonction d'addition dans l'environnement de base :

```
let baselib = Env.add "%add"
                (N (function [ Int a ; Int b ] -> Int (a + b)
                           | _ -> failwith "ERROR")) (* should never happen *)
                Env.empty
```

Et cette fois, le type de l'environnement est `env_value Env.t`, ce qui nous apporte la souplesse nécessaire.

→ Ajoutez les lignes de code nécessaires dans votre fichier `baselib.ml`.

2. Pour pouvoir appeler des fonctions, il nous faut un nouveau type d'expression `Call` qui doit avoir un identifiant de fonction ainsi qu'une liste d'expressions pour les arguments.

   → Ajouter la ligne nécessaire dans le module `Ast`.

3. Il faut maintenant mettre à jour le module `Interp` :
   — utiliser l'environnement de base à la place de l'environnement vide dans `eval` ;
   — faire attention à insérer des valeurs de type `Baselib.env_value` dans l'environnement dans `eval_instr` ;
   — gérer le cas `Call` dans `eval_expr` et penser dans cette même fonction à l'impact de nos modifications sur le cas `Var`.

   → Mettez à jour votre fichier `interp.ml` puis votre fichier `test.ml` pour tester vos modifications.

   Vous pouvez maintenant ajouter d'autres fonctions dans la bibliothèque de base de votre langage. Astuce : regardez la documentation de `List.fold_left`.

Exercice 3.

Mini projet à rendre (contrôle continue)

1. Les mécanismes les plus compliqués ont été couvert dans ce TP.

   Il reste à implémenter :
   — les conditions,
   — les boucles,
   — les définitions de fonctions.

   Pour les conditions et les boucles il n'y a pas de difficultés particulières. Il faut ajouter les cas dans le type `Ast.instr` et puis les gérer dans `Interp.eval_instr`.

   Pour les définitions de fonctions, il faut définir un nouveau type `def` (reportez vous au cours si besoin) et on peut enfin définir le type `prog` comme liste de `def`.

   Il faut aussi définir un nouveau cas dans le type `Baselib.env_value` pour les fonctions définies dans le programme interprétée. Ce cas doit contenir la liste des identifiants des arguments de la fonction et le bloc d'instruction qui compose le corps de la fonction.

   Ensuite il faudra changer la fonction `Interp.eval` (qui a dorénavant pour type `Ast.prog -> Ast.value`) pour qu'elle consiste en l'ajout dans l'environnement de l'ensemble des fonctions définies puis en l'appelle de la fonction `main`.

2. Une fois ces fonctionnalités implémentées, écrivez un programme qui :
   — défini une fonction "guessing_game" qui prend en argument un entier n, choisi un entier x au hasard entre 0 et n, puis fait jouer l'utilisateurice à deviner le nombre x en lui indiquant chaque fois si sa proposition est trop petite ou trop grande ;
   — défini un fonction "main" qui demande à l'utilisateurice un nombre max puis appelle la fonction "guessing_game" avec ce nombre en argument.

   Cela nécessitera bien sûr que vous ayez mis dans votre bibliothèque de base des fonctions de comparaison, des fonctions d'entrées / sorties (`geti`, `puti`, `puts`), et une fonction de génération d'aléa (`rand`).

3. (Bonus) Implémentez les fonctions natives nécessaires à l'exécution du programme écrit en représentation intermédiaire dans le fichier `sierpinsky.ml` fourni avec le cours précédent.