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

Interprétation et compilation

TP 4 : Calculatrice avec Racket

Dans ce TP :
— Écriture d'un interpréteur pour le langage "arith".

Exercice 0.

Interpréteur trivial.

1. Avec le code (corrigé) du dernier TP sur l'analyse sémantique, vous avez un *front-end* tout prêt pour écrire un interpréteur pour un langage d'expression arithmétique et booléenne.
   On va commencer par créer un module pour l'interpréteur et l'appeler depuis le module principal de notre programme à la suite de l'analyse sémantique.
   (a) → Créez un nouveau fichier `eval.rkt`.
   (b) → En vous inspirant de ce qu'on a fait ensemble en cours pour le langage Gawi, faites en sorte que celui-ci fournisse une fonction `interp` qui prend en entrée un AST et sera chargé d'en faire l'interprétation.
   (c) → Dans le module principal, appelez cette fonction en lui passant le résultat de l'analyse sémantique, et affichez le résultat.
2. Au moment où vous commencez votre interprétation, vous avez de déjà de fortes garanties sur votre programme.
   → Lesquelles et pourquoi ?
3. En vous inspirant de la structure de l'analyse sémantique et de ce qu'on a fait ensemble en cours pour l'interprétation du langage Gawi (c'est à dire en suivant la structure de l'AST), écrivez les fonctions suivantes :
   (a) → Une fonction `eval-prog` qui évalue un programme, qui prend en argument l'AST passé à `interp` ainsi qu'un environnement.
   Cette fonction sera appelée par `interp` avec un environnement vide pour l'instant.
   (b) → Une fonction `eval-instr` qui évalue une instruction, qui prend en argument une instruction et un environnement.
   Cette fonction sera appelée par la précédente pour chaque instruction du programme avec l'état actuel de l'environnement. Elle renverra l'état de l'environnement après l'interprétation de l'instruction.
   (c) → Une fonction `eval-expr` qui évalue une expression, qui prend en argument une expression et un environnement.
   Cette fonction sera appelée par la précédente pour évaluer l'expression de l'assignation (qui sont la seule forme d'instruction de notre langage). Elle renverra la valeur de l'expression.
   Pour l'instant, contentez-vous de gérer les cas triviaux c'est à dire les valeurs de type Bool et Num (vous pouvez par exemple renvoyez une erreur dans les autres cas).
4. → Dans la fonction `eval-instr`, affichez pour chaque assignation le nom de la variable et la valeur calculée de l'expression.
5. → Testez votre interpréteur, par exemple avec le fichier de test `exo3-types1.arith` du TP précédent.
   La sortie doit ressembler à quelque chose comme :

```
a: 42
b: #t
```

Exercice 1.

Variables et fonctions.

1. On veut maintenant faire en sorte que le fichier de test `exo3-types2.arith` fonctionne, c'est à dire que `eval-expr` sache prendre en compte les variables.
   → Implémentez cette fonctionnalité et justifiez vos choix.
2. On veut maintenant faire en sorte que le fichier de test `exo3-types3.arith` fonctionne, c'est à dire que `eval-expr` sache prendre en compte les appels de fonction.

(a) → En vous inspirant de ce qu'on a fait en cours pour le langage Gawi, définissez l'environnement initial `*initial-env*` dans le fichier `baselib.rkt`.

(b) → Dans le fichier `eval.rkt`, mettez à jour la fonction `interp` pour qu'elle appelle `eval-prog` avec `*initial-env*` comme environnement inital.

(c) → Mettez à jour la fonction `eval-expr` pour gérer le cas des appels de fonction.

(d) → Testez votre code (avez-vous pensé à l'évaluation récursive des arguments des fonctions ?).

3. On veut maintenant faire en sorte que le fichier de test `exo3-types4.arith` fonctionne, c'est à dire que `eval-expr` sache prendre en compte les conditions.
   → Implémentez cette fonctionnalité et testez votre code.