Pablo Rauzy
pr@up8.edu
https://pablo.rauzy.name/teaching/liec

UNIVERSITÉ PARIS 8 VINCENNES-SAINT-DENIS

Langages : interprétation et compilation

Université Paris 8 – Vincennes à Saint-Denis
UFR MITSIC / L3 informatique

Séance 6 (TP) : Analyse sémantique

N'oubliez pas :
- Les TPs doivent être rendus par courriel au plus tard la veille de la séance suivante avec "[liec]" suivi du numéro de la séance et de votre nom dans le sujet du mail, par exemple "[liec] TP6 Rauzy".
- Quand un exercice demande des réponses qui ne sont pas du code, vous les mettrez dans un fichier texte `reponses.txt` à rendre avec le code.
- Le TP doit être rendu dans une archive, par exemple un tar gzippé obtenu avec la commande `tar czvf NOM.tgz NOM`, où `NOM` est le nom du répertoire dans lequel il y a votre code (idéalement, votre nom de famille et le numéro de la séance, par exemple "rauzy-tp6").
- Si l'archive est lourde (> 1 Mo), merci d'utiliser `https://bigfiles.univ-paris8.fr/`.
- Les fichiers temporaires (si il y en a) doivent être supprimés avant de créer l'archive.
- Le code doit être proprement indenté et les variables, fonctions, constantes, etc. correctement nommées, en respectant des conventions cohérentes.
- Le code est de préférence en anglais, les commentaires (si besoin) en français ou anglais, en restant cohérent.
- **N'hésitez jamais à chercher de la documentation par vous-même sur le net!**

Dans ce TP :
- Écrire un analyseur sémantique de plus en plus complexe.
- Apprendre à rajouter une fonctionnalité à un langage.

Exercice 0.

Mise en place du TP.

1. Pensez à organiser correctement votre espace de travail, par exemple tout ce qui se passe dans ce TP pourrait être dans `~/liec/s6-tp/`.
2. Récupérez les fichiers nécessaires depuis la page web du cours, ou directement en ligne de commande avec `wget https://pablo.rauzy.name/teaching/liec/seance-6_tp.tgz`.
3. Une fois que vous avez extrait le dossier de l'archive (par exemple avec la commande `tar xzf seance-6_tp.tgz`), renommez le répertoire en votre nom (avec la commande `mv liec_seance-6_files votre-nom`). Si vous ne le faites pas tout de suite, pensez à le faire avant de rendre votre TP.
4. Vous êtes encouragé à tester systématiquement votre code après chaque modification de code, cela vous aidera à repérer les erreurs au plus tôt et donc le plus précisément possible.

   Aussi, n'hésitez pas à consulter la documentation de Racket (`raco docs`).

Exercice 1.

État des lieux.

1. On continue d'avancer dans notre petit langage à base d'expression arithmétique. La dernière fois notre analyseur syntaxique acceptait comme programme simplement une expression seule.

   → En vous basant sur les modules du lexer et du parser qui vous sont fournis, donnez une BNF du langage accepté par ceux-ci.
2. → Expliquez les priorités déclarées dans le parseur et leur nécessité (avec des exemples).
3. → Expliquez *brièvement*, ce que font chacune des fonctions définies dans le modules "semantics".

Exercice 2.

Portée des variables.

1. Dans notre petit langage, la portée des variables est simple : à partir du moment où elles sont définies (qu'on leur a assigné une valeur) et jusqu'à la fin du programme.

   (a) → Quelles structures peut-on utiliser pour garder en mémoire les noms des variables déjà définies ?

   (b) → Sachant qu'on va vouloir garder pour chaque variable des informations sur son type, laquelle de ces structures est la mieux adaptées ?

   (c) → Comment appelle-t-on généralement cette structure ?

2. Commençons par la vérification de l'existence dans l'environnement des variables utilisées.
   (a) → Dans le code de l'analyseur sémantique, où devons-nous tester si une variable existe ou non ?
   (b) Il va donc falloir passer en argument à cette fonction un argument additionnel représentant notre environnement.
   → Ajoutez ce paramètre.
   (c) → Rapportez une erreur dans le cas où une variable est utilisée alors qu'elle n'est pas dans l'environnement (n'hésitez pas à consulter la documentation de la fonction `hash-has-key?`)
   (d) → Mettez aussi à jour tous les appels récursifs de la fonction pour lui ajouter son argument supplémentaire.
   (e) → Cette fonction est appelée à un autre endroit, où ?
   (f) → Ajoutez-y l'environnement en argument supplémentaire, et pour qu'il y soit accessible, ajoutez le aussi en argument supplémentaire de la fonction appelante.
   (g) → Même question pour la fonction appelante : mettez à jour l'endroit où elle est appelée et passez en argument l'environnement à la fonction appelante.
   (h) Pour des raisons qu'on verra plus tard lorsqu'on travaillera sur des langages plus complexes, il est important que l'environnement soit "immutable", c'est à dire qu'on ne puisse pas le modifier "sur place" (voir la documentation de `make-immutable-hash`).
   → Dans la fonction `analyze`, passez en argument supplémentaire un environnement initial vide.
3. Pour que ça fonctionne, il faut mettre à jour l'environnement.
   (a) → Dans le code de l'analyseur sémantique, quand peut-on savoir qu'une variable existe ?
   (b) Il faudrait donc pouvoir renvoyer deux valeurs dans la fonction `analyze-instr` : la structure `Assign`, et le nouvel environnement.
   → Comment faire pour renvoyer deux valeurs ?
   (c) → Mettez en œuvre cette méthode. Référez-vous à la documentation de la fonction `cons` pour créer une paire. Référez-vous à la documentation de la fonction `hash-set` pour mettre à jour l'environnement. Pour l'instant on va simplement associer le nom de la variable à la valeur `'defined`.
   (d) Il faut maintenant prendre en compte ce nouveau type de retour dans `analyze-prog`.
   → En utilisant les fonctions `car` (qui récupère le premier élément d'une paire) et `cdr` (qui récupère le second) mettez à jour cette fonction.
4. Vous pouvez tester votre code avec le fichier `err00_unbound-var.arith` qui vous est fourni.

Exercice 3.

Typage.

1. Pour pouvoir tester notre code, nous allons temporairement demander à celui-ci de nous afficher le contenu de l'environnement à la fin de son analyse.
   (a) → Dans la fonction `analyze-prog`, remplacez `null` dans le cas de la liste vide (fin de l'analyse des instructions) par le code suivant (en supposant que votre environnement s'appelle `env`) :

```
(begin
  (when DEBUG
    (hash-for-each env (lambda (v t) (printf "~a: ~a\n" v t))))
  null)
```

   (b) → Quelque part en haut de votre fichier, rajoutez `(define DEBUG #t)`. Vous pourrez bien sûr le mettre à `#f` quand vous n'en aurez plus besoin.
2. Testez votre code sur le fichier `exo3-types1.arith`. Normalement, le débug que l'on vient d'ajouter vous affiche :
   `a: defined`
   `b: defined`
   → Expliquez cet affichage.
3. Notre objectif est que le débug affiche `a: num` et `b: bool`.
   On va donc devoir mettre le type des variables dans l'environnement à la place de `'defined`. Pour cela, il faut que `analyze-expr` nous renvoie le type de l'expression dont on affecte le résultat à la variable.
   → Pour les cas de bases pour l'instant (nombres et booléens), faites en sorte que `analyze-expr` renvoie deux valeurs : l'expression et son type.
4. → Maintenant, mettez à jour la fonction `analyze-instr` pour prendre cela en compte.
5. Testons notre code.
   (a) Retestez votre code avec le fichier `exo3-types1.arith`.
   → Est-ce que ça marche ?

(b) Testez le maintenant avec le fichier `exo3-types2.arith`.
→ Qu'est ce qui échoue cette fois-ci ?

6. → Corrigez le cas où l'expression est une variable. Vous pouvez récupérer la valeur associée à une clef dans une table de hachage avec la fonction `hash-ref`.
7. On veut maintenant que ça marche pour le fichier `exo3-types3.arith`.
→ Lequel des deux cas que nous n'avons pas géré est nécessaire pour le code de ce fichier ?
8. Pour savoir le type de retour d'une fonction, on regarde aussi dans l'environnement. Dans le fichier `stdlib.rkt`, un hash immutable `*types*` est défini. Vérifiez que vous comprenez bien son fonctionnement.
→ Quel est le type de retour de la fonction `%add` ? Quels sont les types attendus de ses arguments ? Mêmes questions pour `%eq` ?
9. → Plutôt que d'initialiser l'environnement avec un hash vide comme on l'a fait à la question 2.h de l'exercice 2, initialisez le avec celui de notre librairie standard.
10. Dans la fonction `analyze-expr` dans le cas d'un appel de fonction, renvoyez son type de retour avec l'expression.
11. On veut maintenant que ça marche pour le fichier `exo3-types4.arith`, c'est à dire renvoyer un type quand une expression est une condition.
Cette fois-ci, le type de l'expression est le type renvoyé par la condition *qui doit donc être le même dans ses deux branches*.
→ Sachant que vous pouvez comparer les types de base avec `eq?`, faites en sorte de renvoyer le type aussi dans le cas d'une expression qui est une condition. Rapportez les erreurs si besoin.
12. Si tout va bien, la variable `e` doit être de type nombre.
→ Testez votre code. C'est bon ?

Exercice 4.

Encore du typage...

1. Il manque encore quelques fonctionnalités importantes dans notre analyse sémantique.
→ Quoi ?
2. En utilisant les fichiers de test `err01_*` à `err08_*`, ajoutez ces deux fonctionnalités.

Exercice 5.

Étendre notre petit langage.

1. Dans cet exercice on veut étendre notre langage pour rajouter la possibilité d'utiliser `&&`, `||`, et `!`, avec leur sémantique habituelle, dans les tests des conditions.
Pour cela, nous devons dans l'ordre :
— les faire reconnaître à l'analyseur lexical,
— adapter la grammaire des expressions dans l'analyseur syntaxique (et les priorités !),
— mettre à jour la librairie standard de notre langage pour ajouter ces fonctions.
→ Est-ce que c'est tout ?
2. Avant de commencer, faisons déjà une petite amélioration.
   (a) → Est-ce qu'on a vraiment besoin des mots clefs `true` et `false` dans notre langage, ("mots clefs" au sens où ils sont gérés dès l'analyse lexicale) ? Si ce n'est pas le cas, comment faire pour s'en passer mais garder leur fonctionnalité ?
   (b) → Faites cette modification.
3. → Au boulot sur l'ajout des opérateurs logiques !
4. → Testez votre code avec le fichier `err09_*`.

Retenez bien cette façon de faire où on étend le langage par "couche" : c'est la bonne approche pour vos projets. Plutôt que de d'abord écrire un lexer énorme, puis un parser énorme, etc, c'est à dire devoir à chaque étape gérer plein de choses d'un coup, il vaut mieux aller d'un bout à l'autre de la chaîne avec très peu de choses (par exemple, juste un seul nombre au début !), et en rajouter petit à petit en faisant toute la chaîne à chaque fois.