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 2 (TP) : Analyse lexicale

N'oubliez pas :
- Les TPs doivent être rendus par courriel au plus tard le lendemain du jour où ils ont lieu avec "[liec]" suivi du numéro de la séance et de votre nom dans le sujet du mail, par exemple "[liec] TP2 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-tp2").
- 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 :
- Apprendre à utiliser l'outil `lex` des `parser-tools` de Racket.

Exercice 0.

Utiliser la documentation locale.

1. La documentation de Racket est particulièrement riche. Cependant, la version en ligne sur `https://docs.racket-lang.org/` correspond à la dernière version stable de la plateforme.
   Racket étant livré avec sa documentation, il est possible d'accéder localement à la documentation de la version effectivement installée sur votre machine.
   → Dans un terminal, lancez la commande `raco docs` qui va ouvrir dans un navigateur la documentation locale de votre installation de Racket.
2. La documentation qui nous intéresse aujourd'hui est celle de l'outil `lex` des `parser-tools`.
   → Rendez-vous dans cette partie de la documentation, et gardez la sous le coude pour la suite du TP.

Exercice 1.

Unités lexicales pour le λ-calcul.

1. À propos du λ-calcul, Wikipédia nous raconte : « *Le lambda-calcul (ou λ-calcul) est un système formel inventé par Alonzo Church dans les années 1930, qui fonde les concepts de fonction et d'application. Il a été le premier formalisme pour définir et caractériser les fonctions récursives et il a donc une grande importance dans la théorie de la calculabilité, à l'égal des machines de Turing.* ».
   En λ-calcul, tout est une fonction. Une *expression* en λ-calcul (ou *λ-terme*) est :
   - soit une *variable* : x, y, foo, bar, ...;
   - soit une *abstraction* : $\lambda x.E$ (où E est un λ-terme);
   - soit une *application* AB (où A et B sont des λ-termes).

   Et c'est tout! On peut avoir besoin de parenthèses pour lever l'ambiguïté sur certains termes : par exemple, le terme $\lambda f.\lambda x.f(fx)$ est différent de $\lambda f.\lambda x.fffx$ qui revient à $\lambda f.\lambda x.(ff)x$ (l'application est associative à gauche).
   → Quelles sont les types d'unités lexicales qu'il va falloir reconnaître ?
2. → Donnez les expressions régulières correspondantes à chacun de ces types.
3. → Dessinez le diagramme de transitions correspondants.
4. → Donnez la table de transition de l'automate fini déterministe correspondant.

Exercice 2.

Premiers lexers en Racket.

1. Ouvrez un nouveau fichier `lexers.rkt` dans lequel vous recopierez ceci :

```
#lang racket/base

(require parser-tools/lex
         (prefix-in : parser-tools/lex-sre))
```

La première ligne déclare qu'on utilise le langage Racket en important seulement le cœur du langage (plutôt que tout plein de modules dont on ne se servira pas).

Le `require` importe le module `lex` des `parser-tools` ainsi qu'un module facilitant l'écriture d'expressions régulières en préfixant ses opérateurs par `:` (pour ne pas qu'ils remplacent des fonctions font on peut avoir besoin comme `+` et `*`).

→ Définissez dans un premier temps un lexer `first-lexer` qui va simplement ignorer les caractères blancs, afficher les autres sur une ligne, et finalement renvoyer le symbole `fini`.

2. → Testez votre lexer pour vérifier qu'il fonctionne correctement et que vous avez bien compris comment s'utilise l'outil, par exemple avec le code suivant :
   `(call-with-input-string "Est-ce que ça marche ?" first-lexer)`
3. → Écrivez un second lexer `second-lexer` sur le modèle du premier mais qui au lieu d'afficher les caractères et de continuer l'analyse les renvoie.
4. → Écrivez une fonction `second-lex` qui prend en argument un port d'entrée et qui utilise le lexer `second-lexer` pour afficher les caractères non-blancs de ce port (comme le faisait `first-lexer`).
5. → Testez à nouveau que ça fonctionne bien :
   `(call-with-input-string "Est-ce que ça remarche ?" second-lex)`
6. → Écrivez maintenant dans votre fichier le code nécessaire pour que votre script Racket puisse prendre en argument un fichier qu'il ouvre en lecture et passe à la fonction `second-lex`.
7. → Testez encore une fois que cela fonctionne bien, cette fois-ci depuis votre terminal :
   `racket lexers.rkt` *`fichier`*, où *`fichier`* est un fichier de votre choix (ça peut être votre `lexers.rkt` si vous n'avez pas d'idée).

Exercice 3.

Un lexer pour le λ-calcul.

1. On va maintenant écrire un vrai analyseur lexical pour le λ-calcul. Ouvrez un nouveau fichier `lc-lexer.rkt` et copiez y la même chose que dans la première question de l'exercice précédent.

   → D'après la documentation, comment déclare-t-on les unités lexicales (tokens) que l'on va faire reconnaître au lexer ?
2. → Déclarez les unités lexicales dont on aura besoin.
3. → Écrivez un lexer `tokenize` qui fait l'analyse lexicale du λ-calcul en utilisant les tokens que vous venez de déclarer (vous pouvez utiliser le symbole `\` ou la chaîne `"lambda"` à la place du symbole `λ`). N'oubliez pas de rapporter les erreurs.
4. → Écrivez une fonction `lex` qui affiche les tokens lus (un par ligne) et si applicable, leur valeur.
5. → Reprenez votre code de l'exercice précédent pour faire l'analyse d'un fichier passé en argument à votre script.
6. → Adaptez maintenant votre lexer pour lui faire retourner automatiquement les positions des tokens en utilisant `lexer-src-pos` à la place de `lexer`. (Attention, vous aurez besoin d'utiliser la fonction `port-count-lines!` pour dire à Racket de suivre les positions des caractères).
   Adaptez aussi la fonction `lex` pour que celle-ci affiche pour chaque token sa position dans le fichier source.
7. → Créez au moins deux fichiers d'exemple (un sans erreur, un avec) pour tester votre code.