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

Interprétation et compilation

TP 1 : Analyse lexicale avec Racket

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 les expressions arithmétiques.

1. Une expression arithmétique est un calcul que l'on peut taper sur une calculatrice par exemple. Dans cet exercice (et dans la suite du TP), on va se limiter aux nombres entiers, et aux quatre opérations de base. On veut aussi avoir la possibilité d'influencer l'ordre d'évaluation des opérations.

   → 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 dont 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 les expressions arithmétiques.

1. On va maintenant écrire un vrai analyseur lexical pour les expressions arithmétiques. Ouvrez un nouveau fichier `arith-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 des expressions arithmétiques en utilisant les tokens que vous venez de déclarer. 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.