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 4 (TP) : Analyse syntaxique

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] TP4 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-tp4").
- 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 `yacc` des `parser-tools` de Racket.

### Exercice 0.

Récupération des fichiers nécessaires.

1. Pensez à organiser correctement votre espace de travail, par exemple tout ce qui se passe dans ce TP pourrait être dans `~/liec/s4-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-4_tp.tgz`.
3. Une fois que vous avez extrait le dossier de l'archive (par exemple avec la commande `tar xzf seance-4_tp.tgz`), renommez le répertoire en votre nom (avec la commande `mv liec_seance-4_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.

La syntaxe du λ-calcul.

1. La λ-calcul a une syntaxe simple. Comme on l'a vu lors du dernier TP, 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).

   Un peu plus formellement on peut décrire la syntaxe du λ-calcul avec la BNF suivante :

   ```
   <expr> ::= <var>
            | "λ" <var> "." <expr>
            | <expr> <expr>
            | "(" <expr> ")"
   <var>  ::= .. un nom de variable valide ..
   ```

   → Identifiez les symboles terminaux, les symboles non-terminaux, ainsi que le symbole de départ de cette grammaire.
2. → Donnez l'ensemble des règles de dérivations de cette grammaire.
3. Étudions d'un peu plus près cette grammaire.
   (a) → S'agit-il d'une grammaire non-contextuelle ? Justifiez.
   (b) → Est-elle ambigüe ou non-ambigüe ? Pourquoi ?

4. → Dans le cas où elle serait ambigüe, proposez-en une version non-ambigüe.

## Exercice 2.

Premiers parsers en Racket.

1. Ouvrez le fichier `parsers.rkt` qui vous a été fourni.

   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 les modules `lex` et `yacc` des `parser-tools`.

   Il y a ensuite un petit analyseur lexical qui est déjà écrit.

   → Quelles sont les unités lexicales reconnues ?

2. Vous allez maintenant écrire votre premier analyseur syntaxique, qui fera usage de cet analyseur lexical.

   Le but de cet analyseur est de vérifier la syntaxe d'un langage où les phrases sont constituées d'une suite de mots et/ou de nombres séparés par des espaces. Les seules réelles contraintes de ce langage sont qu'un mot ne peut contenir que des lettres, et qu'un nombre que des chiffres.

   Par exemple "coucou" et "KdolMp" sont des mots valides, mais pas "dXb87k". De même "421351" et "123" sont des nombres valides mais pas "1234gK67".

   → Donnez une grammaire non-contextuelle et non-ambigüe pour ce langage.

3. → Implémentez l'analyseur syntaxique `parser1` correspondant à cette grammaire en Racket. Il doit renvoyer le symbole `ok` si la syntaxe est valide, et afficher une `Syntax error.` sinon.

4. On voudrait maintenant que l'analyseur syntaxique ne fasse pas que vérifier la syntaxe mais qu'il construise aussi une structure de la phrase.

   On veut un `parser2` qui est similaire à `parser1` sauf qu'il construit une liste des mots et nombres de la phrase. Les mots doivent être des chaînes de caractères et les nombres l'addition de leurs chiffres.

   Par exemple la phrase "coucou 42 pouet lala 13" donnera la liste `("coucou" 6 "pouet" "lala" 4)`.

   → Écrivez `parser2`.

5. On va rajouter un peu de structure à notre langage.

   (a) → Ajoutez des lexèmes vides `Lopar` et `Lcpar` pour les parenthèses et faites en sorte que l'analyseur lexical les reconnaisse.

   (b) → Créez un `parser3` similaire à `parser2` sauf qu'il permet d'ouvrir des "sous-phrases" entre parenthèses. Par exemple "ce super exercice (le 2 (ben oui)) est presque fini (mais après y a le 3 (hé oui (dur hein)) donc ça va)" donnera `("ce" "super" "exercice" ("le" 2 ("ben" "oui")) "est" "presque" "fini" ("mais" "après" "y" "a" "le" 3 ("hé" "oui" ("dur" "hein")) "donc" "ça" "va"))`.

## Exercice 3.

Un parser pour le λ-calcul.

1. → **Rapidement**, réécrivez un lexer pour le λ-calcul (vous pouvez reprendre celui du précédent TP en le nettoyant juste si besoin).
2. → Déclarez les trois structures pour représenter la syntaxe du λ-calcul.
3. → Écrivez un analyseur syntaxique qui construit l'*arbre de syntaxe abstraite* du λ-terme à parser en utilisant les trois structures de la question précédente.
4. → Ajoutez au lexer et au parser ce qu'il faut pour pouvoir rapporter les positions des erreurs.
5. → Conservez aussi les positions (au moins de départ) dans vos structures pour pouvoir signaler des erreurs plus précisément par la suite (quand on fera l'analyse sémantique, par exemple).
6. Bonus → Utilisez `match` (disponible dans le module `racket/match`) pour écrire un pretty-printer récursif pour les λ-termes que vous parsez :
   — pour une variable, afficher son nom ;
   — pour un abstraction, afficher une `(` puis le nom de la variable liée par l'abstraction puis un `.` puis faire l'affichage du corps de l'abstraction puis d'une `)` ;
   — pour une application, faire l'affichage de la fonction puis d'une espace puis celui de l'argument, le tout entre parenthèse.

   Bonus bonus : faire en sorte que les parenthèses inutiles ne soient pas affichées.