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

# Interprétation et compilation

## TP 2 : Analyse syntaxique avec Racket

Dans ce TP :
— Apprendre à utiliser l'outil `yacc` des `parser-tools` de Racket.

### 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 `~/ic/tp2/`.
2. 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 des expressions arithmétiques.

1. Les expressions arithmétiques ont une syntaxe simple que vous connaissez déjà. Comme on l'a vu lors du dernier TP, une expression arithmétique :
   — soit une *valeur* : $13$, $42$, $51$, $123$, ... ;
   — soit une des quatre *opérations* de base en utilisant les opérateurs $+$, $-$, $\times$, et $/$.
   Un peu plus formellement on peut décrire la syntaxe des expressions arithmétiques avec la BNF suivante :

```
<expr> ::= <num>
         | <expr> "+" <expr>
         | <expr> "-" <expr>
         | <expr> "*" <expr>
         | <expr> "/" <expr>
         | "(" <expr> ")"
<num>  ::= "-" ? [ "0"-"9" ] +
```

   → 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, comment peut-on la désambigüer ?

### Exercice 2.

Un premier parser en Racket : les expressions arithmétiques.

1. → Rapidement, réécrivez un lexer pour les expressions arithmétiques (vous pouvez reprendre celui du précédent TP en le nettoyant juste si besoin).
2. → Déclarez les cinq structures pour représenter la syntaxe des expressions arithmétiques.
3. → Écrivez un analyseur syntaxique qui construit l'*arbre de syntaxe abstraite* d'une expression à parser en utilisant les cinq 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 expressions arithmétiques que vous parsez :
   — pour une nombre, simplement l'afficher ;
   — pour une opération, afficher une `(` puis son opérande de gauche puis le symbole lui correspondant puis l'opérande de droite puis une `)`.

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

## Exercice 3.

Une grammaire alternative pour les expressions arithmétiques.

1. On appelle “polonaise inversée” la notation suffixe. Par exemple, ce qu’on note habituellement en notation infixe `3 + 4` s’écrit `3 4 +` en notation polonaise inversée.

   Autre exemple : `3 + 4 * 2` s’écrit `3 4 2 * +` ou `4 2 * 3 +`.

   → Donnez une grammaire pour les expressions arithmétiques avec cette notation.
2. → Que remarquez-vous par rapport à la notation infixe habituelle ?
3. → Quelle structure de données vous semble adaptée pour représenter cette syntaxe ?
4. → En réutilisant le lexer de l’exercice précédent, écrivez un parser pour les expressions arithmétiques en notation polonaise inversée.
5. Bonus → En utilisant une structure de pile (vous pouvez simplement utiliser une liste pour représenter votre pile), écrivez un évaluateur pour les expressions arithmétiques en notation polonaise inversée que vous avez parsé.

   Bonus bonus : que remarquez-vous ?