# TP 2 : Introduction à Yacc

## 1 Prise en main de Yacc : parenthésage correct

Le but de cet exercice est de développer une grammaire pour un parenthésage correct d'une expression. Les expressions sont très simples (et pas forcément utiles) : elles sont composées de constantes (des entiers), il y a un seul opérateur binaire `+`, mais il y a trois types de parenthèses : `()`, {} et `[]`. Les parenthèses doivent être équilibrées et correspondre, et une expression formée avec l'opérateur `+` ne doit pas être ambiguë (associativité, voir aussi § 2), il faut donc entièrement parenthéser les additions imbriquées, avec n'importe quel type de parenthèse.

*A faire :*

1. Développez une grammaire pour ce langage, et calculez la somme des constantes contenues dans l'expressions.
2. Étendez la grammaire et surtout les actions sémantiques, pour calculer la somme des constantes, et le nombre de paires pour chaque type de parenthèses. Il peut être utile de définir des fonctions auxiliaires dans la section initiale du fichier Yacc (entre `%{ ... %}`).

*Exemples* (première ligne : entrée, deuxième ligne : réponse du programme si l'expression est bien formée) :

— `(1) + {2}`
`Sum: 3 Parens: 1 Braces: 1 Square brackets: 0`
expression bien formée

— `{[(3) + {{4}}]}`
`Sum: 7 Parens: 1 Braces: 3 Square brackets: 1`
expression bien formée

— `(1 + {2}`
expression mal formée, parenthèses non équilibrées

— `(1} + {2)`
expression mal formée, parenthèses ne correspondent pas

— `1 + 2 + 3`
expression mal formée parce que ambiguë. Des parenthésages corrects seraient : `(1 + 2) + 3` ou `1 + {2 + 3}` ou aussi `[{1} + (2 + 3)]`.

## 2 Analyse d'ambiguïtés

Les exercices suivants ont pour but de vous familiariser avec des conflits qui peuvent apparaître dans des parsers LALR (et dont fait partie la famille des parser Yacc) : *shift-reduce* et *reduce-reduce* et qui indiquent souvent une ambiguïté de la grammaire. Certains problèmes sont dûs à un *lookahead* insuffisant des parsers LALR, bien que la grammaire soit parfaitement bien définie. Nous développons des méthodes de réécrire les grammaires pour éviter ces problèmes.

L'invocation avec l'option `-v` génère un fichier qui permet d'analyser des problèmes. Par exemple, `ocamlyacc -v parser.mly` génère `parser.output`.

## 2.1 Reduce/Reduce Conflict

Intuitivement, un tel conflit apparaît quand plusieurs règles peuvent être utilisées pour dériver le même mot, par exemple dans une grammaire comme celle-ci :

```
start: nt1 B
     | nt2 B ;
nt1: A;
nt2: A;
```

Après avoir vu le symbole `A` et avant de lire `B`, le parser ne sait pas s'il faut construire un arbre de dérivation avec `nt1` ou `nt2`. Dans ce cas, il rapporte un conflit, par exemple :

```
42: reduce/reduce conflict (reduce 2, reduce 3) on B
state 42
nt1 : A .  (2)
nt2 : A .  (3)
```

Ici, les numéros se réfèrent à la numérotation des règles (voir début du fichier `parser.output`). Typiquement, ocamlyacc résout le conflit en appliquant l'une des règles, ce qui peut avoir pour conséquence que l'autre règle n'est jamais utilisée ("rules never reduced").

*A faire :* Analysez la grammaire sur les conflits Reduce/Reduce qui permet d'analyser des expressions qui apparaissent dans un langage comme Python, où l'opérateur de multiplication peut s'appliquer entre expressions arithmétiques (comme `3 * 4`) ou aussi en "multipliant" un entier avec une liste, (comme `3 * [4]`).

## 2.2 Shift/Reduce Conflict : Associativité et Priorité

Intuitivement, un tel conflit apparaît quand on peut ou bien continuer à analyser le mot actuel avec une règle $r_1$ (*shift*) ou terminer l'analyse d'une sous-expression à l'aide d'une autre (ou la même ...), disons $r_2$, et construire un nouveau sous-arbre avec $r_2$ (*reduce*). Supposons que nous nous intéressons aux expressions arithmétiques $e$ et $r_1$ est $e : e + e$ et $r_2$ est $e : e * e$. Sans avoir défini des priorités pour les opérateurs $+$ et $*$, on peut analyser $1 * 2 + 3$ de deux manières différentes. Après avoir lu $1 * 2$, on peut effectuer un *reduce* avec $r_2$, ce qui correspondrait au parenthésage $(1 * 2) + 3$, ou un *shift* du symbole $+$, ce qui continue l'analyse de l'expression $2 + 3$ avec la règle $r_1$ et correspondrait au parenthésage $1 * (2 + 3)$.

*A faire :*

1. Développez une grammaire pour des expressions arithmétiques composées, d'abord, uniquement de nombres et de l'opérateur binaire $+$. Quelques expressions valides de ce langage sont `1 + 2` et `1 + 2 + 3`. En quoi consiste l'ambiguïté identifiée par Ocamlyacc ? Par quel mécanisme pouvez-vous obtenir une associativité à gauche, sans introduire explicitement des parenthèses ?
2. Rajoutez l'opérateur $*$. Comme évoqué plus haut, la grammaire est maintenant ambiguë à cause d'un problème de priorité. Etudiez comment ceci se manifeste dans les messages d'Ocamlyacc. Modifiez la grammaire en introduisant de nouvelles règles et des parenthèses pour obtenir l'interprétation usuelle des expressions.

   Les actions sémantiques de vos règles devraient afficher les expressions entièrement parenthésées.

   *Exemple :*

   ```
   > 1 + 2 * 3
   (1+(2*3))
   > 1 * 2 + 3 * 4
   ((1*2)+(3*4))
   > 1 * (2 + 3) * 4
   ((1*(2+3))*4)
   ```

3. En (1) et (2), on vous demande de définir l'associativité et priorité avec un codage explicite par des règles. Vous pouvez aussi utiliser des déclarations de priorité (voir transparents p. 15, lisez aussi la section Declarations du manuel).

### 2.3 Shift/Reduce Conflict : *If-then-else*

Nous étudions un langage impératif avec deux genres d'instructions $s$, à savoir
— une affectation de la forme $v = n$, où $v$ est une variable et $n$ est un nombre (pour faire simple), et
— un conditionnel de la forme `if` $n$ `then` $s$ `else` $s$, où $n$ est un nombre (sémantiquement, le choix de la branche `then` pourrait se faire pour $n \neq 0$, mais c'est sans importance ici).
— un conditionnel sans `else`, donc de la forme `if` $n$ `then` $s$

*A faire :*

1. Développez une grammaire qui reconnaît ce langage. Analysez le problème mis en exergue par Ocamlyacc et expliquez-le à l'aide de l'instruction `if 1 then if 2 then x = 42 else x = 43`.
2. Documentez-vous sur la manière dont le problème est typiquement résolu (par exemple en C) [1] et implantez une grammaire qui évite le conflit.

## 3 Grammaire pour formules logiques

Le but de l'exercice suivant est d'écrire une grammaire pour des formules de la logique du 1er ordre.

### 3.1 Formules propositionnelles bien parenthésées

Implantez la grammaire des formules bien parenthésées de la logique propositionnelle, c.-à.-d. des formules composées de
— *cas de base :* variables propositionnelles (des identificateurs qui commencent avec des majuscules)
— *cas récursif unaire :* la négation de formules : `(~`$F$`)`,
— *cas récursif binaire :* la conjonction, disjonction, implication de formules : `(`$F$ `/\` $G$`)`, `(`$F$ `\/` $G$`)`, `(`$F$ `-->` $G$`)`

où $F$ et $G$ sont des formules.

*Exemple :* `(~ ((A1 --> A2) \/ ((~ B) /\ C)))` est une formule bien formée selon cette grammaire.

### 3.2 Formules du 1er ordre

On introduit d'abord des *termes*. Les termes sont
— *cas de base :* des variables de terme (des identificateurs qui commencent avec des minuscules)
— *cas récursif :* des termes composés $f$`(` $t_1$ `,` $\ldots$`,`$t_n$ `)`, où $f$ est une variable de terme et les $t_i$ sont des termes. Attention : la liste des sous-termes peut être vide.

On étend maintenant la définition des formules :
— *cas de base :* au delà des variables propositionnelles, on admet des prédicats $P$`(` $t_1$ `,` $\ldots$`,`$t_n$ `)`, où $P$ est une variable propositionnelle et les $t_i$ sont des termes. Encore une fois, la liste des termes peut être vide.
— *cas récursifs :* au delà des opérateurs unaire et binaires, on introduit les quantificateurs `!!` (pour $\forall$) et `#` (pour $\exists$). Les formules quantifiées ont la forme `(` `!!` $x_1 \ldots x_n$ `.` $F$ `)` (et pareil pour `#`), où $x_1 \ldots x_n$ est une liste non vide de variables propositionnelles et $F$ une formule.

*Exemple :* `(!! x y .  ( P(f(x,y)) /\ (# z. ~ R(z, c()))))` est une formule bien formée.

**Important :** codez correctment
— les listes non vides (dans ce cas : de variables)
— les listes avec sépérateur (dans ce cas : de termes)

1. https://fr.wikipedia.org/wiki/Dangling_else

Il est utile de se rappeler les définitions récursives : Ainsi, une liste non vide de variables est ou bien une variable, ou bien une variable suivi d'une liste non vide de variables.

### 3.3 Opérateurs avec priorité

Dans la section 3.1, les opérateurs Booléens ont été introduits par priorité décroissante : `~` a la priorité la plus forte, `-->` la priorité la plus faible. Coder cette priorité permet d'omettre quelques parenthèses. Ainsi, la formule `(~ ((A1 --> A2) \/ ~ B /\ C))` est maintenant bien formée, et sa syntaxe abstraite est identique à celle de la formule de la section 3.1.