

Addendum: Grammaires LL(2)

Martin Odersky

15 novembre 2005

version 1.3

Problèmes avec la grammaire Zwei

Si l'on considère ce sous-ensemble de la grammaire Zwei :

```
Statement = Formal "=" Expression
           | ident "=" Expression
           | Expression
  Formal   = Type ident
    Type   = ident
           | ...
  Expression = ident
           | ...
```

Lorsqu'un lexème `ident` est lu, le prochain lexème peut être :

- un autre `ident` : c'est une déclaration de variable ;
- un `=` : c'est un assignement de variable ;
- autre chose : c'est une expression.

C'est-à-dire qu'à ce moment, on ignore quelle est l'alternative qui est en train d'être reconnue.

Plus généralement, on peut montrer que :

- la **grammaire** pour Zwei n'est pas LL(1) — ce qui est assez commun : Scala, Java ou C# sont dans le même cas ;
- la **grammaire** pour Zwei est LL(2) ;
- le **language** Zwei est LL(1), c'est à dire qu'il existe une autre grammaire pour Zwei qui est LL(1).

Ces deux dernière propriétés amènent à deux solutions pour utiliser quand même une analyse par descente récursive pour Zwei :

- 1 Modifier l'analyseur syntaxique pour accepter les grammaire LL(2) : c'est la solution que nous recommandons.
- 2 Transformer la grammaire de Zwei en une grammaire LL(1) par factorisation sur la gauche : c'est possible mais assez laborieux.

Analyse syntaxique d'une grammaire LL(2)

Un analyseur syntaxique LL(2) requiert l'ajout à l'analyseur lexical d'une fonction **def peekAhead: Token** qui :

- retourne le lexème **suivant** le lexème courant.
- ne modifie pas la valeur de **token**, **chars** et **pos** dans l'analyseur lexical.

Rendre un analyseur lexical compatible LL(2)

Pour rendre un analyseur lexical compatible avec une grammaire LL(2), les modifications suivantes sont requises :

- 1 Définir une variable pour stocker toute l'information obtenue par `peekAhead` :

```
var pushedBack: Option[Triple[Int,String,Token]] = None;
```
- 2 Modifier `nextToken` comme suit :

```
def nextToken: Unit = pushedBack match {  
  case Some(Triple(p, c, t)) =>  
    token = c; chars = c; pos = p;  
    pushedBack = None;  
  case None =>  
    ... // comme precedement  
}
```
- 3 Ajouter la méthode `peekAhead` elle-même.

Implantation de `peekAhead`

```
def peekAhead: Token = pushedBack match {  
  case Some(Triple(p, c, t)) => t  
  case None =>  
    val savedToken = token;  
    val savedChars = chars;  
    val savedPos = pos;  
    nextToken;  
    pushedBack = Some(Triple(pos, chars, token))  
    token = savedToken;  
    chars = savedChars;  
    pos = savedPos;  
    pushedBack.get._3  
}
```

Rendre un analyseur syntaxique compatible LL(2)

Finalement, à l'aide de la méthode `peekAhead`, la partie critique du code de `Statement` dans l'analyseur syntaxique peut être écrit de la manière suivante :

```
def Statement = {  
  if (token == IDENT) {  
    peekAhead match {  
      case IDENT =>  
        Formal; accept(EQUALS); Expression  
      case EQUALS =>  
        nextToken; accept(EQUALS); Expression  
      case _ =>  
        Expression  
    }  
  }  
  else ...  
}
```