

# Compilation

## Analyse syntaxique

PAR DAVID TELLER

[David.Teller@univ-orleans.fr](mailto:David.Teller@univ-orleans.fr)

Au cours du TD précédent, nous avons développé, à l'aide de FLEX, un analyseur lexical pour notre mini-langage mathématisant :

```
function f : (x,y,a,b) ->
  when x < y -> a
  when x >=y -> b
end
```

```
function g : (x) ->
  print(f(x,1-x,0,1))
end
```

Nous allons progressivement nous servir de cet analyseur lexical pour construire un analyseur syntaxique. Pour vous aider, je vous ai déjà tapé le squelette de cette analyseur syntaxique, accompagné d'un Makefile. Vous trouverez le tout sur ma page web (rubrique *Compilation* de <http://www.univ-orleans.fr/lifo/Members/David.Teller/teaching.html>, en anglais dans le texte, parce que nous le valons bien).

## Examen du fichier

Le fichier `compilateur.parser.yy` définit une *grammaire* pour Bison, associée à quelques actions, et qui doit permet de reconnaître le langage – sans rien en faire d'intéressant pour le moment.

**Exercice 1.** Ajoutez des `printf` partout, à l'image de ce qui est déjà fait, histoire de pouvoir suivre le déroulement de l'analyse syntaxique. Que remarquez-vous lorsque vous appliquez `compilateur < test.toy` ?

**Exercice 2.** Pourquoi cette définition bizarre de `expression_0`, `expression_1...` et pas une définition en une seule fois de la forme suivante ?

```
expression : expression GREATER expression
           | expression GREATEREQ expression
           ...
           | OPEN expression CLOSE
```

## Calculs

Le fichier `test_expressions.toy` contient quelques fonctions bidon qui devrait vous permettre de tester vos capacités de calcul mental. Ou, si possible, celles de l'ordinateur. Tel est l'objectif de cette section.

Vers le début du fichier `compilateur.parser.yy`, vous pouvez apercevoir les lignes :

```
%type <expression_value> expression_3
...
%type <expression_value> expression
```

Le rôle de ces directives est de préciser qu'une `expression` a pour résultat une valeur de type `expression_value`, lui-même défini un peu plus haut dans le code source. De même,

```
%token <float_value>  NUM
%token <string_value>  NAME
```

précise qu'un lexème `NUM` a toujours une valeur de type `float_value`, alors qu'un lexème `NAME` a toujours une valeur `string_value`.

**Exercice 3.** Adaptez les règles `expression`, `expression_0` ... de manière à ce que, lorsqu'elles rencontrent effectivement des nombres ou des opérations sur des nombres, (et non pas des noms de variables ou de fonctions), elles effectuent effectivement les calculs. Pour ce faire, inspirez-vous de la règle `expression_3`.

Débrouillez-vous pour que la règle `expression` affiche le résultat des calculs en question.

**Exercice 4.** Ajoutez le traitement du moins unaire (c'est-à-dire reconnaître que `MINUS 1` signifie probablement  $-1$ ).

## Arbre de syntaxe abstraite

Notre objectif, avec cet analyseur syntaxique, n'est ni d'afficher des messages pas drôles au fur et à mesure de l'exécution, ni de tenir le rôle de simple calculatrice. Nous cherchons plutôt à transformer le flux de lexèmes en quelque chose de plus simple à manipuler.

La grammaire définie dans le fichier `compilateur.parser.yy` représente une *syntaxe concrète*, c'est-à-dire la syntaxe telle que va la voir l'utilisateur du langage. À l'inverse, le développeur du langage manipulera une *syntaxe abstraite*, plus simple, et qui ne se préoccupe plus de la priorité des opérateurs, de détails comme la présence ou l'absence de virgules, etc.

Le fichier `compilateur.ast.h` définit une partie de la syntaxe abstraite, celle concernée par `expression`.

**Exercice 5.** Complétez la définition de la syntaxe abstraite, à l'aide d'un type pour les `when`, un type pour les définitions de fonctions et un type pour le programme lui-même.

**Exercice 6.** Complétez le fichier `compilateur.parser.yy` pour qu'il renvoie un arbre de syntaxe abstraite conforme à ce que vous venez d'ajouter dans `compilateur.ast.h`

## Pretty-printer

Un pretty-printer est un programme qui, à partir d'un arbre de syntaxe abstraite ou concrète, fabrique un code source.

**Exercice 7.** Écrivez une fonction (ou, plus probablement, plusieurs fonctions mutuellement récursives), qui, à partir d'un arbre de syntaxe abstraite, retrouveront à peu de choses près le code source originel.

Comme les espaces, les retours à la ligne et le format des nombres ont été oubliés, vous êtes libres de faire comme bon vous semble à leur sujet.

**Exercice 8.** Combinez le résultat de l'exercice 6 et celui de l'exercice 7 pour que votre compilateur prenne un code source et renvoie un code source.