

Grammaires et Analyse Syntaxique - Cours 11

Études de cas : langages de données et de programmation

Ralf Treinen



`treinen@irif.fr`

15 avril 2022

Regard sur quelques langages

- ▶ Nous allons réviser des langages de programmation et langages de données que vous avez (éventuellement) vu pendant votre cursus, sous l'aspect de
 - ▶ Analyse syntaxique et construction d'un arbre de syntaxe abstrait ;
 - ▶ Typage statique ou dynamique, pour les langages de programmation.

Typage statique et typage dynamique

- ▶ Typage statique : vérification des types après construction de la syntaxe abstraite, et *avant* l'exécution.
- ▶ Avantage : Sûreté. Une fois le typage vérifié par le compilateur on sait que des erreurs de *type* ne peuvent plus se produire (sauf dans certains langages de programmations qui permettent de contourner le typage).
- ▶ Avantage : Le compilateur peut annoter la syntaxe abstraite par des types des expressions, ce qui simplifie l'exécution. Désambiguïsation des opérateurs surchargés.
- ▶ Exemples : Java, OCaml, ...
- ▶ Il y a aussi des inconvénients, dont nous parlerons plus tard.

Typage statique et typage dynamique

- ▶ Typage dynamique : vérification des types pendant l'*exécution* du programme, quand on applique un opérateur à des valeurs.
- ▶ Avantage : plus de flexibilité (mais : flexibilité aussi possible dans le cas statique, par ex. polymorphie).
- ▶ Avantage : programmes moins verbeux, car il n'est pas nécessaire de déclarer les variables avec leur type (mais : le même avantage peut être obtenu par une *inférence de type*).
- ▶ Inconvénient : l'interpréteur doit garder *pendant l'exécution* l'information de type de toutes les valeurs.
- ▶ Inconvénient : les erreurs de typage sont découvertes pendant l'exécution.
- ▶ Exemples : Python, bash, Perl.

Le format XML

- ▶ Extensible Markup Language
(fr. : *langage de balisage extensible*)
- ▶ Balise : lexème (un mot, un symbole) qui indique la structure dans un document. Par exemple des parenthèses.
- ▶ Objectif : un langage simple et universel pour l'échange de données structurées, à la fois lisible pour des humains et des machines.
- ▶ On peut critiquer que le langage est trop verbeux pour être les humains, et plutôt conçu pour un traitement automatique.
- ▶ Il s'agit d'une instance d'un format plus général, du nom SGML (**S**tandard **G**eneralized **M**arkup **L**anguage).
- ▶ Le langage XML est standardisé par le W3C (World Wide Web. Consortium).

La structure d'un document XML

- ▶ Un document XML décrit un arbre.
- ▶ Un nœud peut avoir un nombre arbitraire (0 inclus) d'enfants.
- ▶ Un nœud peut aussi avoir des *attributs*.
- ▶ Les enfants d'un nœud sont appelés ses *éléments*; un élément peut être un morceau de texte, ou un autre nœud.

Exemple

```
<menu id="file" value="File">  
  <popup>  
    <menuitem value="New" onclick="CreateNewDoc()">  
  </menuitem>  
    <menuitem value="Open" onclick="OpenDoc()">  
  </menuitem>  
    <menuitem value="Close" onclick="CloseDoc()">  
  </menuitem>  
  </popup>  
</menu>
```

(sans utiliser le raccourci pour les nœuds sans enfants)

Remarques

- ▶ Ça ressemble à du HTML, et pour cause : XML et HTML sont tous les deux des dérivés du langage plus général SGML.
- ▶ L'utilisation des chevrons < et > est assez caractéristique pour ces langages.
- ▶ Le XML est plus stricte que le HTML, par exemple :
 - ▶ XML exige que les valeurs des attributs sont écrites entre guillemets, contrairement aux premières versions de HTML.
 - ▶ XML exige que toutes les balises sont proprement fermées, contrairement à HTML qui est très libéral en ce regard.

Vers une définition de la syntaxe

- Un nœud commence avec une balise (angl. : *tag*) de la forme

`<m>`

et se termine avec

`</m>`

m est un mot quelconque, appelé le *nom* de cet élément.

- Le nom est obligatoirement le même dans la balise de début et de fin du même nœud. Il est permis d'utiliser le même nom pour plusieurs nœuds du même arbre.
- Une balise de début peut en plus contenir des paires d'attributs avec leurs valeur.

Vers une définition de la syntaxe

- Une paire attribut/valeur est donnée dans la forme

attr=valeur

où *attr* est un mot (appelé l'attribut), et *valeur* est une chaîne de caractères entre guillemets (appelée la valeur).

- Il n'est pas permis définir dans la même balise deux fois le même attribut.

Exemple : Données OpenStreetMap

```
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6"
  copyright="OpenStreetMap and contributors"
  license="http://opendatacommons.org/licenses/odbl/1-0/">
  <way id="62378611"
    visible="true" version="8" changeset="20691652"
    timestamp="2014-02-21T11:23:38Z" user="thibdrev" uid="1279506">
    <nd ref="779143878"/>
    <nd ref="2198721646"/>
    <nd ref="2198721727"/>
    .....
    <tag k="amenity" v="university"/>
    <tag k="building" v="yes"/>
    <tag k="name" v="Halle aux Farines (Université Paris Diderot)"/>
    <tag k="source" v="cadastre-dgi-fr source : Direction Générale des
    <tag k="wheelchair" v="yes"/>
    <tag k="wikipedia" v="fr:Université Paris VII - Diderot"/>
  </way>
</osm>
```

Une grammaire pour XML ?

- ▶ Terminaux (jetons) : `mot`, `string`, `texte`, `/`, `=`, `<`, `>`, `</`
- ▶ Non-terminaux : `Arbre`, `Start`, `End`, `Attrbs`, `Elements`
- ▶ Axiome : `Arbre`
- ▶ Règles :

`Arbre` → `Start Elements End`

`Start` → `< mot Attrbs >`

`End` → `</ mot >`

`Attrbs` → ϵ | `mot = string Attrbs`

`Elements` → ϵ | `Arbre Elements` | `texte Elements`

Une grammaire pour XML ?

- ▶ Attention : Cette grammaire ne définit pas complètement le langage XML.
- ▶ Il y a deux éléments de la définition qui ne sont pas exprimés par la grammaire :
 - ▶ Toute balise de début doit être fermée par une balise de fin *du même nom*.
 - ▶ Tous les attributs dans une balise de début doivent être *différents*.
- ▶ Ces deux restrictions ne peuvent pas être réalisées dans la grammaire, à cause du lemme d'itération.

Validation du XML

- ▶ Il est inévitable que la grammaire définit seulement une *sur-approximation* du langage XML.
- ▶ Il nous faut une étape supplémentaire de validation qui vérifie les deux restrictions supplémentaires : l'analyse *sémantique*.
- ▶ Au moins on peut construire un arbre de syntaxe abstraite sans vérifier les balises, et faire l'étape de vérification des balises sur l'arbre de syntaxe abstraite.
- ▶ On rencontre le même problème dans les langages de programmation : toute variable doit être déclarée avant son utilisation.

Le format YAML

- ▶ **YAML : Yaml Ain't Markup Language** (YAML n'est pas un langage de balisage)
- ▶ JSON et XML utilisent des balises pour indiquer la structure du document.
- ▶ YAML est un format assez riche, un document YAML peut même contenir des morceaux de JSON.
- ▶ Nous regardons ici un seul aspect de YAML : l'utilisation de l'incrémentation des lignes pour indiquer une structure (similaire au langage de programmation *Python*).

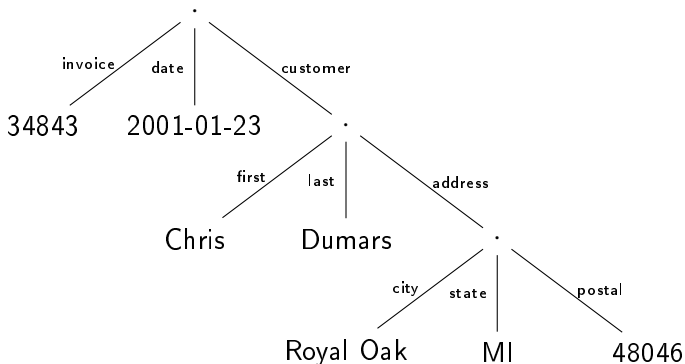
Un fragment de YAML

- ▶ Nous nous intéressons au fragment de YAML qui permet de définir des arborescences.
- ▶ On souhaite représenter par un document en YAML un arbre dont les arêtes sont étiquetées par des chaînes de caractères, et dont les feuilles portent des chaînes de caractères.
- ▶ Contrainte supplémentaires : toutes les arêtes partant du même nœud portent des étiquettes différentes.

Exemple d'un document YAML

```
invoice : 34843
date    : 2001-01-23
customer :
  first  : Chris
  last   : Dumars
  address:
    city   : Royal Oak
    state  : MI
    postal : 48046
```

Représentation sous forme d'un arbre



Comment lire un tel document YAML

- ▶ Une ligne peut contenir une paire clef – valeur.
- ▶ Une ligne peut aussi contenir seulement une clef, dans ce cas une sous-structure doit suivre à partir de la ligne suivante.
- ▶ L'incrémentation des lignes indique le début ou la fin d'une sous-structure.

Comment faire l'analyse syntaxique ?

- ▶ Premier problème : il faut assurer que les clefs sont différentes pour chaque nœud dans l'arborescence.
- ▶ Problème assez similaire à la vérification des balises dans un document XML : on peut d'abord construire l'arbre, et puis faire cette vérification sur l'arbre construit.
- ▶ Deuxième problème : comment analyser les différences dans l'incrémentation des lignes, qui jouent ici le rôle de parenthèses ?
- ▶ On aura le même problème avec le langage Python, voir plus tard !

Bash

- ▶ Analyse syntaxique à la volée (intercalée avec l'exécution).
- ▶ Un mot peut être un mot clé ou pas selon l'endroit où il paraît.
- ▶ Possibilité de faire interpréter une chaîne de caractères comme une commande de la shell (commande `eval`).
- ▶ Typage dynamique : bash ne construit pas d'arbre de syntaxe abstraite avant l'exécution, il n'y a donc aucun moyen pour analyser statiquement le typage.

Bash : analyse syntaxique à la volée

```
#!/bin/bash
```

```
# les deux premières lignes sont exécutées avant
```

```
# que l'erreur de syntaxe est détectée.
```

```
echo "hello"
```

```
echo "bonjour"
```

```
if fi
```

Bash : si un mot est un mot clef dépend du contexte

```
#!/bin/bash
```

```
for do in for do in echo done; do echo $do; done
```

- ▶ le premier do est le nom d'une variable
- ▶ le deuxième do est une valeur du type string
- ▶ le troisième do est un mot clef

Bash : eval

```
#!/bin/bash  
# évaluation d'une chaîne construite pendant  
# l'exécution du programme.
```

```
i=10
```

```
mot="if _(( _i_>_0_ )); _then_ _echo_ 'if ' ; _(( i-- )); _fi "  
eval $mot
```

```
mot=$(sed -e 's/if/while/g' -e 's/then/do/' \  
          -e 's/fi/done/' <<< $mot)  
eval "$mot"
```


Bash : typage et évaluation

- ▶ Il n'y a pas de types, les valeurs des variables sont des chaînes caractères.
- ▶ Il y a une syntaxe distincte pour les expressions arithmétiques.
- ▶ Si on veut des listes on les encode dans des string, avec les éléments séparés par des espaces, ce qui est une source riche de bugs.
- ▶ Modèle d'exécution bizarre qui confond les booléens avec la présence/absence d'erreurs d'exécution.

Python

- ▶ Analyse syntaxique complète avant exécution.
- ▶ Syntaxe : la structure du programme est indiquée par le niveau d'*incrément* des lignes : pas un langage algébrique.
- ▶ Typage dynamique, typage statique seulement par des outils externes (par exemple `mypy`).

Python : analyse syntaxique complète

```
#!/usr/bin/python3  
# erreur de syntaxe detectee avant execution  
  
print("hello")  
print("bonjour")  
if fi
```

Syntaxe de Python

```
if x < 0:  
    x = x-1  
else:  
    x = x+1  
    x = x*x
```

a une *structure différente* de

```
if x < 0:  
    x = x-1  
else:  
    x = x+1  
x = x*x
```

Erreur : changement de l'incrémentation dans le même bloc

```
#!/usr/bin/python3
```

```
if x > 42 :
```

```
    print(x)
```

```
    print(x+x)
```

```
    print(x*x)
```

Erreur : retour à un niveau d'incrémentement inconnu

```
#!/usr/bin/python3
```

```
print(32)  
if _x_>_17_  
    print(x)  
    if _x_>_42_  
        print(x+x)  
    print(x*x)
```

Comment reconnaître les niveaux d'incrémentations ?

- ▶ Idée : utiliser des jetons OPEN et CLOSE, envoyés par l'analyse lexicale quand l'incrémentations change.
- ▶ L'analyse lexicale maintient une *pile* de niveaux d'incrémentations actuellement actives, initialisée à une valeur 0.
- ▶ Quand l'analyse lexicale trouve une incrémentations de ligne augmentée : envoie le jeton OPEN, empile la nouvelle valeur.
- ▶ Quand l'analyse lexicale trouve une incrémentations de ligne réduite : envoie le jeton CLOSE, supprime la valeur de la pile.
- ▶ Il y a un problème dans le dernier cas ...

Problème avec la fermeture d'un niveau

```
#!/usr/bin/python3
```

```
x=42
if _x_>_1_:
    print("niveau_1")
    if _x_>_10_:
        print("niveau_2")
    print("niveau_0")
```

On peut fermer plusieurs niveaux sur un seul coup !

Solution

- ▶ Quand plusieurs niveaux sont fermés : il faut en principe envoyer plusieurs jetons `CLOSE` à la fois!
- ▶ Or, on attend de l'analyse lexicale un seul jeton à la fois.
- ▶ Solution : L'analyse lexicale maintient un compteur pour les fermetures de niveaux qui restent à communiquer.
- ▶ Voir le code dans le répertoire `mini-python` pour les détails.

Python : typage dynamique

```
#!/usr/bin/python3  
# la meme variable peut prendre des valeurs de  
# type different.
```

```
x=42  
print(x)  
x=True  
print(x)
```

Python : typage dynamique

```
#!/usr/bin/python3  
# erreur de typage seulement detectee quand  
# la branche else est executee
```

```
import sys  
while True:  
    read=sys.stdin.readline()  
    if read != 'coocoo\n':  
        print(17+21)  
    else:  
        print ( 'abc ' & 'def ' )
```

Java

- ▶ Construction de la syntaxe abstraite par le compilateur.
- ▶ Typage statique.
- ▶ L'information de typage doit être écrite explicitement dans le programme.
- ▶ Vérification statique (donc, par le compilateur) des exceptions.
- ▶ Inconvénient : les programmes sont verbeux.

OCaml

- ▶ Typage statique.
- ▶ *Inférence* de type : le compilateur infère les types des identificateurs, il n'est pas nécessaire de les écrire dans le programme.
- ▶ On peut écrire dans le programme les types des identificateurs si on le souhaite.
- ▶ Avantage : programmes concis, et en même temps sûreté de typage.
- ▶ Inconvénient : l'inférence de type limite les possibilités de surcharge des opérateurs.
- ▶ Syntaxe critiquable pour l'absence de mots clef qui terminent des expressions complexes.

OCaml : absence de mots terminateurs

```
type t = A of t | B of t | C of t | I of int
```

```
let f x = match x with  
| A y -> true  
| B y -> match y with  
| A z -> false  
| B z -> true  
| C z -> false  
| I n -> true  
| C y -> false  
| I n -> true
```