TD : Lexing et Parsing — semaine 1

4 février 2021

Objectif(s)

- ✯ Rappel du fonctionnement d'un analyseur lexical et d'un parseur.
- ✯ Élaboration d'un compilateur complet pour un langage simple appelé Calc.
- ✯ Étude de différentes transformation de grammaire vers des grammaires *LL(1)*.

Exercice 1 – Les expressions régulières

1. Lister les différentes phases de la compilation et comment elles interagissent.
2. Donner l'expression régulière des expressions arithmétiques avec les opérateurs + et * ainsi que les chiffres (exemple : `3+3*2`).
3. Est-il possible d'écrire une expression régulière pour ce même langage mais avec des parenthèses pour grouper des sous-expressions (exemple : `(3+3)*2`)?

Exercice 2 – La limite de l'analyse lexicale : commentaires imbriqués

Les commentaires en Objective CAML sont hiérarchiques. On peut ainsi commenter des parties de texte, y compris celles contenant des commentaires. Un commentaire commence par les caractères (* et se termine par *). Voici un exemple :

```
(* début du commentaire
      sur plusieurs
         lignes *)

let succ x = (* fonction successeur *)
  x + 1;;

(* niveau 1 texte commenté
  let old_succ y = (* niveau 2 fonction successeur niveau 2 *)
    y +1;;
    niveau 1 *)
  succ 2;;
```

1. Est-ce qu'une expression régulière peut reconnaitre ce langage ?

   Les lexers proposent une extension permettant de changer d'automate en cours de route grâce à du code arbitraire dans les actions sémantiques. Par exemple :

```
{NUMBER} [0-9]
{LETTER} [a-zA-Z]
{DASH} -

%x automate2
%%
{NUMBER} { BEGIN(automate2); }

<automate2> {LETTER} { BEGIN(0); }
<automate2> {DASH} {}
```

va rentrer dans l'automate `automate2` seulement après avoir lu un chiffre ; `0` est l'indice de l'automate initial. Cet exemple ne reconnait pas le mot `0` dans la phrase `a0`, mais tous les mots dans la phrase `0-a0`.

2. Utilisez cette extension pour reconnaitre les commentaires imbriqués.

Exercice 3 – Un évaluateur arithmétique complet

Cet exercice a pour but de réviser comment les différentes parties d'un compilateur interagissent. Nous allons concevoir un évaluateur simple mais complet pour un langage arithmétique, appelé Calc, contenant les opérateurs `+`, `*`, les nombres entiers ainsi que des expressions entre parenthèses. Le programme renverra le résultat de l'évaluation ; par exemple : `9+2*(2+1)` doit renvoyer `15`.

Vous trouverez quelques notes sur `OCamllex` et `Menhir` en fin de sujet.

1. En séparant bien dans différents fichiers tous les composants d'un compilateur, donnez le code en OCaml du compilateur du langage Calc.
2. Quelle est la différence majeure entre un compilateur écrit en OCaml (ou dans un style fonctionnel) et un compilateur écrit en Java (ou dans un style orienté-objet) ?
3. Réfléchissez et listez les différents composants supplémentaires qu'un compilateur devrait prendre en compte pour être utilisable en pratique.

Exercice 4 – Transformation de grammaire

1. Ajouter le support des opérateurs `-` et `/`, des identifiants et des appels de fonction à la grammaire Calc. On appellera cette grammaire Calc++.
2. Si on utilise un parseur descendant qui produit une dérivation à gauche de cette grammaire (aka. grammaire *LL(1)*), on va boucler dans `expr`, vu que la règle s'appelle récursivement sans rien lire. Donner la grammaire sans récursivité à gauche (indice : pensez à ce que vous feriez si vous deviez programmer ce parseur à la main).
3. Certaines grammaires ont la propriété de pouvoir être implémentée par un parseur qui reconnaît n'importe quelle phrase du langage sans jamais faire d'erreur (aka *backtrack-free*). C'est à dire que lorsque le parseur est face à plusieurs alternatives (séparées par `|`), il choisit toujours la bonne du premier coup. Les grammaires dites *LL(1)* ont cette propriété si le parseur regarde, au plus, à un token en avance dans le flux. On peut rendre certaine grammaire *LL(1)* en factorisant à gauche la grammaire. Cette transformation factorise les préfixes communs entre plusieurs alternatives, par exemple :

   ```
   R = 'a' R2
     | 'a' R3
   ```

   devient

   ```
   R = 'a' R1
   R1 = R2 | R3
   ```

   Effectuer cette transformation sur la grammaire Calc++.

Exercice 5 – Grammaire ambigüe : le cas du IF THEN ELSE

Parfois, certaines grammaires sont ambigües : pour une même phrase, le parseur peut produire plusieurs AST différent. Un exemple de grammaire ambigüe est le traitement des IF-THEN-ELSE sans délimiteur, par exemple :

`IF c THEN IF c THEN b ELSE b`

où on utilise c pour représenter une condition et b pour le bloc de code.

1. Donner une grammaire pour ce IF-THEN-ELSE et montrer qu'elle peut produire deux ASTs distincts pour l'exemple précédent.
2. En Pascal, ce problème est résolu en interdisant un IF sans ELSE dans la première branche d'un IF-ELSE. Donner la grammaire correspondante.

Notes sur OCamllex

Un fichier d'entrée de ocamllex se compose de :

```
{
  (* entête - expression caml - même chose qu'en Menhir *)
  open Parser
}

let alpha = ['a'-'z']*
let eol = '\n'

rule tokens = parse
    alpha            { ALPHA }
  | eol              { EOL   }
  | ['0' - '9'] as i { INT (int_of_string i) }
```

Ici, les caractères doivent être compris entre deux quotes (par exemple `'s'`) et une chaîne entre deux double-quotes (par exemple `"toto"`). Ce qui donne : `['a' - 'z']` pour dire tous les caractères entre le `a` minuscule et le `z` minuscule.

Notes sur Menhir

Un fichier d'entrée de Menhir se compose de 4 parties :

```
%{
(* (1) entête - expression OCaml *)
%}

(* (2) déclaration *)

%token EOL
%token<string> ALPHA
%start main
%type <(String, String)> main

%%
(* (3) règle_i  action_i *)

main : a1=ALPHA EOL a2=ALPHA { (a1, a2) } ;

%%
(* (4) queue - expression caml *)
```

A partir du fichier d'entrée, Menhir génère un fichier OCaml. On trouve textuellement en tête de ce dernier la partie (1) et en queue la partie (4). En général :

— (1) contient les ouvertures de modules (`open`) utilisées par les actions sémantiques contenues dans (3).
— Les lexèmes sont déclarés en (2) ainsi que la déclaration de l'axiome et des types de l'AST généré par les différentes règles.