

Grammaires et Analyse Syntaxique - Cours 2 Générateurs d'analyse lexicale : ocamllex Grammaires (partie 1)

Ralf Treinen



treinen@irif.fr

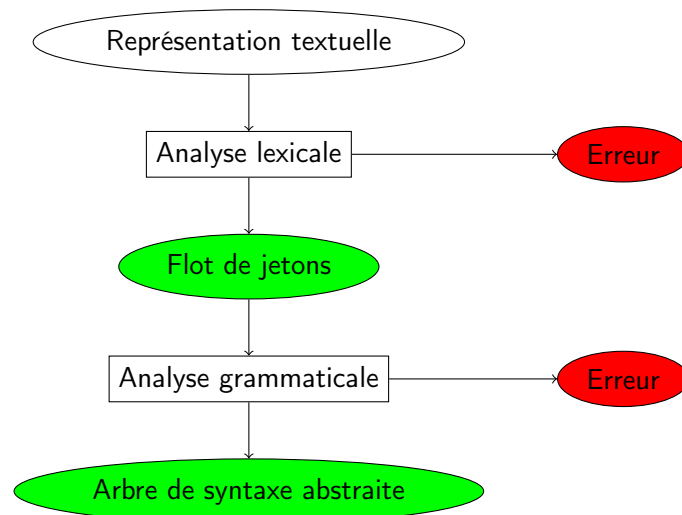
1^{er} février 2024

© Ralf Treinen 2020–2024

Objectif de l'analyse syntaxique

- ▶ Deux objectifs :
 - ▶ Détecter des textes d'entrée qui ne sont pas correctement formés (et donner des indications utiles sur la nature de l'erreur).
 - ▶ Si l'entrée est correcte, construire un arbre de syntaxe.

Découpage de l'analyse syntaxique



Pourquoi deux étapes séparées ?

- ▶ On essaye de faire autant d'analyses que possible dans l'analyse lexicale.
- ▶ Raison : l'exécution d'un automate est très efficace (temps linéaire dans la longueur du texte d'entrée).
- ▶ Problème : l'expressivité des automates est limitée : théorème de l'étoile, théorème de Myhill-Nerode.
- ▶ On verra aujourd'hui un cas concret qu'on ne peut pas reconnaître avec les expressions rationnelles.

L'interface de l'analyseur lexicale

- ▶ On pourrait imaginer que l'analyse lexicale va créer une liste OCaml avec tous les jetons créés lors de l'analyse.
- ▶ Problème : cette liste risque d'être très longue.
- ▶ Normalement, la phase suivante de l'analyse a seulement besoin de lire les jetons au fur et à mesure.
- ▶ Pour ces raisons, l'analyse lexicale crée les jetons un après l'autre *à la demande*.
- ▶ L'interface du code engendré contiendra une fonction qui permet d'obtenir le jeton suivant (ou de signaler la fin de l'entrée).

Un exemple complet d'un fichier de spécification

```
{
  open Token
  exception Lexing_error of string
}
let space=[' '\n\t']
let letter=['A'-'Z' 'a'-'z']
let digit=['0'-'9']

rule next_token = parse
| "(" { PARG }
| ")" { PARD }
| "+" { PLUS }
| "*" { MULT }
| digit+ { INT(int_of_string(Lexing.lexeme lexbuf)) }
| letter(letter|digit)* { ID(Lexing.lexeme lexbuf) }
| space* { next_token lexbuf }
| eof { EOF }
| _ { raise (Lexing_error (Lexing.lexeme lexbuf)) }
```

Fichier de spécification pour ocamllex

- ▶ On écrit un fichier de spécification ocamllex dont le nom se termine sur `.mll`.
- ▶ Ce fichier a (normalement) 3 parties :
 1. une entête (angl. : *header*) entre accolades { et }
 2. une séquence de définitions d'expressions rationnelles
 3. plusieurs (souvent un seul) *points d'entrée* de l'analyse lexicale. Chacun point d'entrée regroupe une séquence d'expressions rationnelles, avec les actions associées.
 4. Optionnellement une partie de code final entre accolades { et }

La première partie : l'entête

- ▶ Contient du code OCaml entre accolades { et } qui est copié au début du fichier (module) engendré
- ▶ Peut être vide (il faut quand même écrire les accolades)
- ▶ Utile par exemple pour :
 - ▶ Définir des fonctions qui vont être utilisées dans les actions
 - ▶ Ouvrir des autres modules qui contiennent des définitions utilisées dans les actions
 - ▶ Définir des variables modifiables, table de hachage, etc. qui vont être manipulées dans les actions.

La deuxième partie : définitions d'expressions rationnelles

- ▶ Définir des abréviations pour des expressions rationnelles
- ▶ Cette partie peut être vide
- ▶ Exemple :
`let letter = ['A'-'Z''a'-'z']`

La troisième partie : Les points d'entrée

- ▶ De la forme générale :

```
rule entrypoint =  
  parse regexp { action }  
  | ...  
  | regexp { action }  
and entrypoint =  
  parse ...  
and ...
```

- ▶ chacun des *entrypoint* est un identificateur OCaml
- ▶ chacune des *regexp* est une expression rationnelle
- ▶ chacune des *action* est une expression OCaml

Constructions d'expressions rationnelles

- ▶ " *string* " : une chaîne de caractères
- ▶ `_` : exactement un caractère
- ▶ `eof` : fin de l'entrée
- ▶ classe de caractères entre `[` et `]`
- ▶ complément d'une classe de caractères entre `[^` et `]`
- ▶ des raccourcis préalablement définis
- ▶ opérateurs post-fix `*`, `+`, `?`
- ▶ juxtaposition de deux expressions : concaténation
- ▶ union de deux expressions : opérateur `|`
- ▶ parenthèses (`(` et `)`)

Plus sur les points d'entrée

- ▶ Souvent un seul point d'entrée
- ▶ Les points d'entrée peuvent avoir des arguments (rarement utilisé)
- ▶ Le code engendré contient pour chaque point d'entrée une fonction du même nom. Si le point d'entrée a n arguments, la fonction engendrée à $n + 1$ arguments, le dernier étant `lexbuf` qui est le flot d'entrée (un type abstrait)
- ▶ La fonction va
 - ▶ lire du flot d'entrée le lexeme (mot le plus long décrit par une des expressions rationnelles);
 - ▶ évaluer l'expression *action* associée, et renvoyer sa valeur.
 - ▶ S'il y a plusieurs cas pour le lexeme lu : on prend le premier.
 - ▶ L'action peut être un appel à la même fonction (demande de lire le lexeme suivant).

Plus sur les actions

- ▶ On a accès au lexeme trouvé par `Lexing.lexeme lexbuf`
- ▶ Si on veut ignorer le lexeme et continuer l'analyse lexicale : l'action est simplement un nouvel appel à l'entrypoint (voir l'exemple)
- ▶ On peut aussi dans l'expression rationnelle donner un nom à une sous-expression rationnelle, et dans l'action associée utiliser ce nom pour le sous-mot trouvé. Par exemple :

```
| "[" ([0-9]+ as number) "]"  
  { BRACKETINT(int_of_string number)}
```

si on veut reconnaître un entier écrit entre des crochets, et mettre dans le jeton associé la valeur de cet entier.

Retour à notre exemple complet : arith.mll

```
{  
  open Token  
  exception Lexing_error of string  
}  
let space=[' '\n\t']  
let letter=['A'-'Z' 'a'-'z']  
let digit=['0'-'9']  
  
rule next_token = parse  
| "(" { PARG }  
| ")" { PARD }  
| "+" { PLUS }  
| "*" { MULT }  
| digit+ { INT(int_of_string(Lexing.lexeme lexbuf)) }  
| letter(letter|digit)* { ID(Lexing.lexeme lexbuf) }  
| space* { next_token lexbuf }  
| eof { EOF }  
| _ { raise (Lexing_error (Lexing.lexeme lexbuf)) }
```

La quatrième partie (si présente)

- ▶ contient du code OCaml qui est copié par ocamllex à la fin du fichier engendré.
- ▶ parfois utile pour éviter d'écrire un programme principal séparé.

Le fichier interface : arith.mli

```
(* le module d'analyse lexicale *)  
  
(* fonction pour demander le jeton suivant *)  
val next_token: Lexing.lexbuf -> Token.token  
  
(* cas d'une erreur lexicale *)  
exception Lexing_error of string
```

Le fichier token.ml

```
type token =
| INT of int
| ID of string
| PARG
| PARD
| MULT
| PLUS
| EOF

let to_string = function
| INT(n) -> "INT("^(string_of_int n)^")"
| ID(s) -> "ID("^(s)^")"
| PARG -> "PARG"
| PARD -> "PARD"
| MULT -> "MULT"
| PLUS -> "PLUS"
| EOF -> "EOF"
```

Le fichier readarith.ml

```
let ch = open_in (Sys.argv.(1)) in
let lb = Lexing.from_channel ch
in
try
  while true do
    let t = Arith.next_token lb
    in
    Printf.printf "%s\n" (Token.to_string t);
    if t=Token.EOF then exit 0
  done
with
Arith.Lexing_error s ->
Printf.printf "Unexpected character: %s\n" s
```

Le fichier token.mli

```
(* module of tokens for arithmetic expressions *)

(* type of tokens *)
type token =
  INT of int | ID of string
  | PARG | PARD | MULT | PLUS | EOF

(* conversion to string *)
val to_string : token -> string
```

Comment compiler ?

- ▶ À la main (ou dans un Makefile) :
 1. ocamllex arith.mll engend arith.ml
 2. ocamlc token.mli engend token.cmi
 3. ocamlc -c token.ml engend token.cmo
 4. ocamlc arith.mli engend arith.cmi
 5. ocamlc -c arith.ml engend arith.cmo
 6. ocamlc -c readarith engend readarith.cmo
 7. ocamlc token.cmo arith.cmo readarith.cmo engend a.out
- ▶ Avec ocamlbuild : ocamlbuild readarith.native
- ▶ Avec dune : dune build, mettre dans le fichier dune :
(ocamllex (modules arith))

Multiple points d'entrées

- ▶ Parfois utile si on veut avoir des règles d'analyse lexicale dans des contextes différents.
- ▶ Correspond à la technique des états multiples de jflex
- ▶ Tout point d'entrée est traduit par ocamllex en une fonction OCaml.
- ▶ Exemple : lire un fichier d'entiers. Les commentaires entre "(" et ")" doivent être ignorés (comme en OCaml)

Le problème avec la première tentative

- ▶ Le programme va, sur une entrée contenant *plusieurs* commentaires, sauter toute la partie de l'entrée entre le début du premier commentaire et la fin du dernier commentaire.
- ▶ La raison est qu'on cherche toujours le mot *le plus long* qui est filtré par une des expressions régulières.
- ▶ Or ici, exceptionnellement, on veut pour le cas du commentaire le mot le plus court !
- ▶ Solution : utiliser deux points d'entrée !

Première tentative

```
{
  open Token
  exception Lexing_error of string
}
let space=[' '\n'\t']
let digit=['0'-'9']

rule next_token = parse
| digit+ { INT(int_of_string(Lexing.lexeme lexbuf)) }
| space* { next_token lexbuf }
| "(" _* ")" { next_token lexbuf }
| eof { EOF }
| _ { raise (Lexing_error (Lexing.lexeme lexbuf)) }
```

Version correcte

```
{
  open Token
  exception Lexing_error of string
  exception Lexing_eof_in_comment
}
let space=[' '\n'\t']
let digit=['0'-'9']

rule next_token = parse
| digit+ { INT(int_of_string(Lexing.lexeme lexbuf)) }
| space* { next_token lexbuf }
| "(" { token_after_comment lexbuf }
| eof { EOF }
| _ { raise (Lexing_error (Lexing.lexeme lexbuf)) }
and token_after_comment = parse
| ")" { next_token lexbuf }
| eof { raise Lexing_eof_in_comment }
| _ { token_after_comment lexbuf }
```

Mots clés d'un langage de programmation

Solution naïve : une règle par mot clefs.

```
{
  open Token
  exception Lexing_error of string
}
let space=[' '\n'\t']
let letter=['A'-'Z''a'-'z']
let digit=['0'-'9']

rule next_token = parse
| "begin" { BEGIN }
| "end" { END }
| "class" { CLASS }
| digit+ { INT(int_of_string(Lexing.lexeme lexbuf)) }
| letter(letter|digit)* { ID(Lexing.lexeme lexbuf) }
| space* { next_token lexbuf }
| eof { EOF }
| _ { raise (Lexing_error (Lexing.lexeme lexbuf)) }
```

Comment reconnaître les mots clés sans catégories dédiées ?

- ▶ Problème : avec une règle par mot clef l'automate peut devenir trop grand.
- ▶ Message d'erreur de ocamllex : *ocamllex : transition table overflow, automaton is too big*
- ▶ Dans presque tous les langages de programmation, tous les mots clés sont des séquences de lettres en minuscules.
- ▶ Solution : Mettre une seule règle pour les identificateurs et les mots clés.
- ▶ Dans l'action associée, on cherche (par ex. dans une table de hachage) si le lexeme est un mot clef, et on crée un jeton en fonction.

Attention à l'ordre des règles

- ▶ Entrée : `beg begin beginner`
- ▶ Premier appel à `next_token` : seulement la cinquième règle s'applique \Rightarrow token ID.
- ▶ Deuxième appel à `next_token` : les règles (1) et (5) s'appliquent au même lexeme `begin`, c'est donc la première parmi ces deux qui gagne \Rightarrow token BEGIN.
- ▶ Troisième appel à `next_token` : les règles (1) et (5) s'appliquent mais la dernière reconnaît un lexeme plus long \Rightarrow token ID.

Le fichier `arith.mll`

```
{
  open Token
  exception Lexing_error of string

  let keyword_table = Hashtbl.create 3;;
  List.iter (fun (kwd, tok) -> Hashtbl.add keyword_table kwd tok)
    [ ("begin", BEGIN);
      ("end", END);
      ("class", CLASS) ]
}
let space=[' '\n'\t']
let letter=['A'-'Z''a'-'z']
let digit=['0'-'9']

rule next_token = parse
| digit+ { INT(int_of_string(Lexing.lexeme lexbuf)) }
| letter(letter|digit)* as id
  { try Hashtbl.find keyword_table id
```

Le fichier arith.mll II

```
with Not_found -> ID id }  
| space* { next_token lexbuf }  
| eof { EOF }  
| _ { raise (Lexing_error (Lexing.lexeme lexbuf)) }
```

Introduction : Grammaires

- ▶ Vu jusqu'à maintenant : analyse lexicale (ocamllex)
- ▶ Nous commençons à présenter le formalisme utilisé pour l'étape d'analyse grammaticale qui suit l'analyse lexicale : les *grammaires* (suite la semaine prochaine).
- ▶ Dans les semaines à venir nous allons étudier la mise en œuvre de l'analyse grammaticale.

Pour savoir tout sur ocamllex

- ▶ Documentation ocamllex :
<https://v2.ocaml.org/manual/lexyacc.html>
Seulement les deux premières sections 17.1 et 17.2 sont pertinentes car nous n'allons pas utiliser ocaml yacc (nous avons mieux!)
- ▶ Documentation du module Lexing :
<https://v2.ocaml.org/api/Lexing.html>

Langages non-reconnaissables

- ▶ On a vu dans le cours AAL3 du L2 deux méthodes pour montrer qu'un langage donné n'est pas reconnaissable (c-à-d ne peut pas être défini par une expression rationnelle) :
 - ▶ Le lemme de l'étoile (ou lemme d'itération, *pumping lemma*)
 - ▶ Le théorème de Myhill-Nerode
- ▶ Intuitivement : les langages qui nécessitent un compteur non borné, ne sont pas reconnaissables.
- ▶ Exemple : $\{a^n b^n \mid n \geq 0\}$ n'est pas reconnaissable (vu dans le cours AAL3).
- ▶ On verra aujourd'hui que ce langage peut être décrit par une grammaire.

Définition des grammaires algébriques

Une *grammaire algébrique* (ou *grammaire hors contexte*) est un tuple $G = (\Sigma, N, S, P)$ où

- ▶ Σ est un ensemble fini de symboles, appelé les *symboles terminaux* de G ;
- ▶ N est un ensemble fini et disjoint de Σ de symboles, appelé les *symboles non-terminaux* de G ;
- ▶ $S \in N$, appelé l'*axiome* de G ;
- ▶ P est un ensemble fini de *règles de production* de la forme $A \rightarrow u$, où $A \in N$, et $u \in (\Sigma \cup N)^*$.

En anglais : *context free grammar (CFG)*.

Premier exemple d'une grammaire algébrique

$G_1 = (\Sigma, N, S, P)$ où

- ▶ $\Sigma = \{a, b\}$
- ▶ $N = \{A\}$
- ▶ $S = A$
- ▶ P consiste en les règles suivantes :

$$A \rightarrow aAb$$

$$A \rightarrow \epsilon$$

- ▶ On verra que cette grammaire définit le langage $\{a^n b^n \mid n \geq 0\}$.

Notation : alternatives

- ▶ Une grammaire peut avoir plusieurs règles avec le même côté gauche :

$$E \rightarrow E+E$$

$$E \rightarrow E * E$$

- ▶ Nous permettons dans la suite dans ce cas d'écrire une seule règle, avec plusieurs *alternatives* sur le côté droit :

$$E \rightarrow E+E \mid E * E$$

Exemple : Dérivation de $aaabbb$ dans G_1

La grammaire G_1 donnée au-dessus a les deux productions :

$$A \rightarrow aAb \mid \epsilon$$

Elle permet par exemple la dérivation suivante :

$$A \rightarrow aAb$$

$$\rightarrow aaAbb$$

$$\rightarrow aaaAbbb$$

$$\rightarrow aaabbb$$

Dérivation en une étape

Définition

Soit une grammaire $G = (\Sigma, N, S, P)$, et $u, v \in (N \cup \Sigma)^*$.

On dit que G permet de dériver v à partir de u en une étape, noté $u \rightarrow_G v$ (ou abrégé $u \rightarrow v$), si

1. $u = w_1 A w_2$, où $w_1, w_2 \in (N \cup \Sigma)^*$, $A \in N$;
2. $A \rightarrow w$ est une règle de P ;
3. $v = w_1 w w_2$.

Dérivation en plusieurs étapes

Définition

Soit une grammaire $G = (\Sigma, N, S, P)$, et $u, v \in (N \cup \Sigma)^*$.

On dit que G permet de dériver v à partir de u en plusieurs étapes, noté $u \rightarrow_G^* v$ (ou abrégé $u \rightarrow^* v$), s'il existe une suite finie w_0, w_1, \dots, w_n de mots de $(N \cup \Sigma)^*$ telle que

$$u = w_0 \rightarrow_G w_1 \rightarrow_G \dots \rightarrow_G w_n = v$$

Exemple

- ▶ $A \rightarrow_{G_1}^* aaabbb$
- ▶ $A \rightarrow_{G_1}^* aaaaaabbbbbbb$
- ▶ $A \rightarrow_{G_1}^* \epsilon$

Langage engendré

Définition

Soit $G = (\Sigma, N, S, P)$ une grammaire algébrique. Le langage engendré par G est

$$\mathcal{L}(G) = \{w \in \Sigma^* \mid S \rightarrow_G^* w\}$$

Un langage L est algébrique (ou hors-contexte, en anglais context free) s'il existe une grammaire algébrique G telle que $L = \mathcal{L}(G)$.

Exemple

$$\mathcal{L}(G_1) = \{a^n b^n \mid n \geq 0\}$$