

# Grammaires et Analyse Syntaxique - Cours 2

## Générateurs d'analyse lexicale : ocamllex

Ralf Treinen



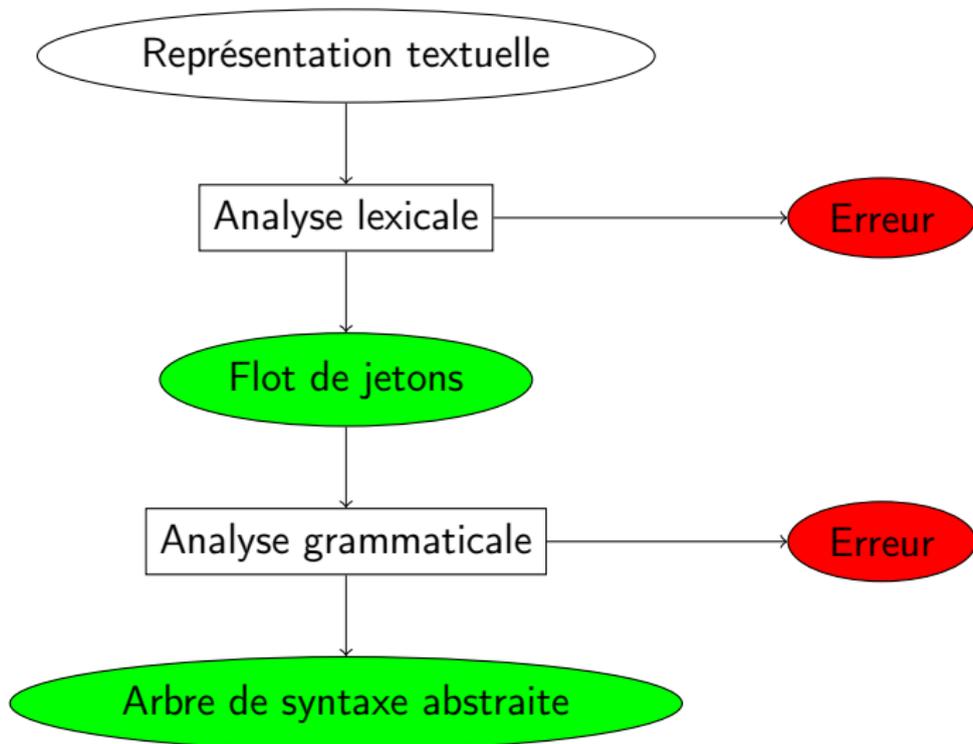
Université  
Paris Cité



`treinen@irif.fr`

29 janvier 2025

## 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 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).

## 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 }

## 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)) }
```

## 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']

## 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 )

## 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

## 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.

## 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.

## 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)) }
```

## 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 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
```

## 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
```

## 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)

## 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)) }
```

## 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 !

## 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 clefs 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)) }

```

## 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.

## Comment reconnaître les mots clefs 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 clefs sont des séquences de lettres en minuscules.
- ▶ Solution : Mettre une seule règle pour les identificateurs et les mots clefs.
- ▶ 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.

## 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
            [ ("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)) }
```

## Pour savoir tout sur ocamllex

- ▶ Documentation ocamllex :  
<https://v2.ocaml.org/manual/lex yacc.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>