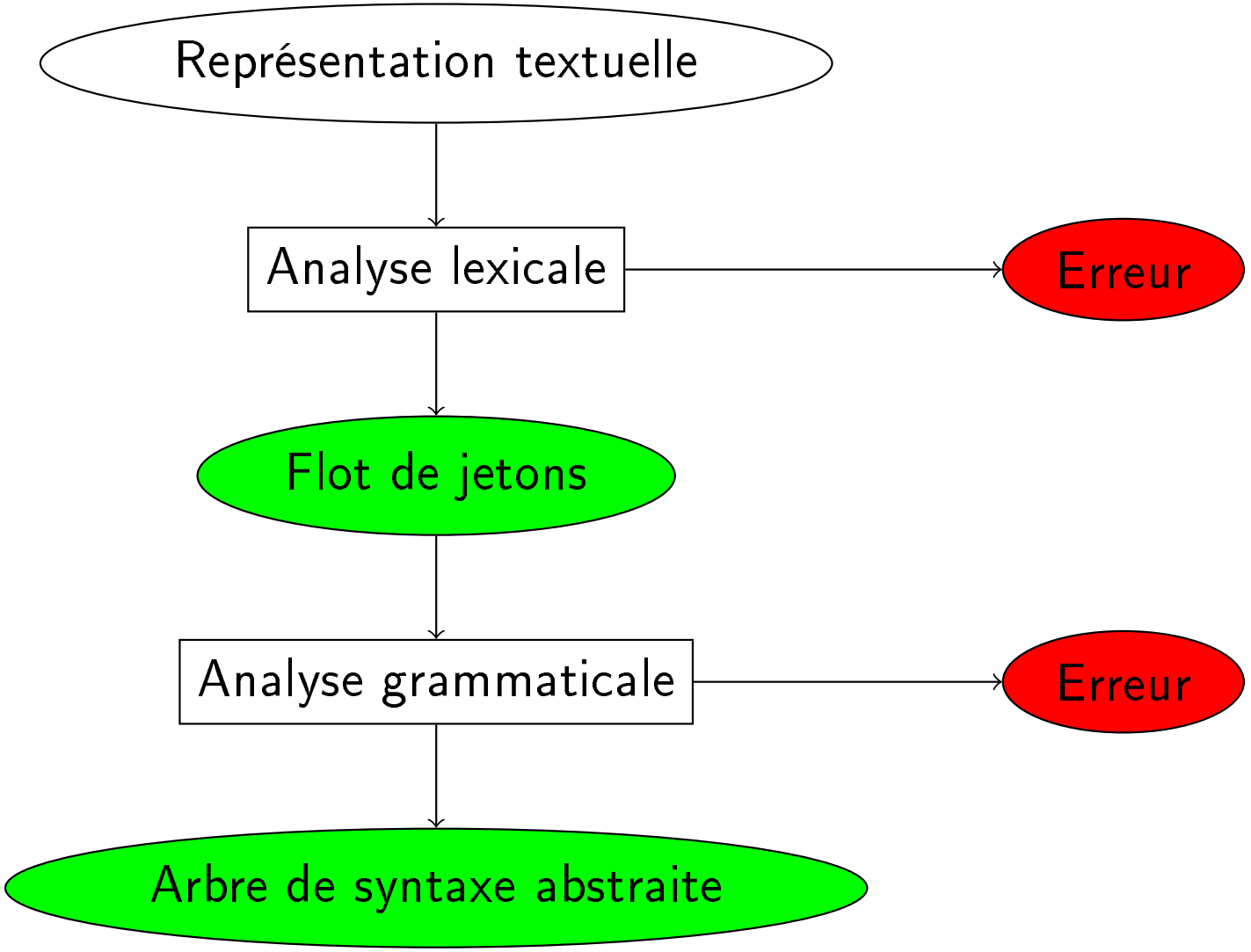
Grammaires et Analyse Syntaxique - Cours 2
Générateurs d'analyse lexicale

Ralf Treinen

treinen@irif.fr

28 janvier 2022

Découpage de l'analyse syntaxique



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.
- Focalisons d'abord sur le premier objectif : distinguer les textes corrects des textes incorrects.

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 la semaine prochaine.

Rappel : le rôle de l'analyse lexicale

- Lire un flot de caractères (typiquement d'un fichier)
- Découper le flot en *lexemes*
- Produire un flot de *jetons* (angl : token)
- Exemple :
  - Flot de caractères en entrée :

(	7	5	6	e	2		*		(	e	5	e	7		+		v	a	l	e	u	r	2	)	)

  - Flot de jetons en sortie :

    PARG INT MULT PARG IDENT PLUS IDENT PARD PARD

Spécification de l'analyse lexicale

- Dans un premier temps on peut imaginer une spécification, qui consiste en
  1. une séquence d'expressions rationnelles
  2. pour chaque expression rationnelle une action qui est soit
     - retourner un certain jeton
     - ignorer la partie de l'entrée lue, et continuer (cas des espaces)
- En vérité on aura besoin de plus de flexibilité dans les actions.
- Exemple (incomplet) :

[0..9]+(e[0..9]+)?	retourner INT
[a..zA..Z][a..zA..Z0..9]*	retourner IDENT
+	retourner PLUS
[ ]*	ignorer et continuer

Première possibilité : programmer à la main

- Il est possible de coder l'analyse lexicale à la main
- Pour cela on utilisera ce qu'on a appris en AAL3 :
  - La traduction des expressions rationnelles en automates
  - L'implémentation des automates
  - Complication : on ne cherche pas à accepter ou rejeter un mot, mais à *découper* un flot d'entrée
  - Émettre un jeton chaque fois qu'on a reconnu une partie de l'entrée
- Possible mais répétitif, et on risque de faire des erreurs.

Deuxième possibilité : utiliser un générateur de code

- Les premiers compilateurs étaient écrits complètement à la main.
- Par exemple le compilateur du langage FORTRAN en 1957 : 18 personnes-années de travail.

  - John Backus, auteur principal de FORTRAN et de son compilateur
  - *Turing Award* 1977

- L'alternative est de faire *engendrer* des analyseurs lexicale et syntaxique à partir d'une spécification.
  C'est la technique utilisée pour l'écriture des compilateurs modernes.

Publication de l'équipe de Backus en 1957

The FORTRAN Automatic Coding System

J. W. BACKUS†, R. J. BEEBER†, S. BEST‡, R. GOLDBERG†, L. M. HAIBT†,
H. L. HERRICK†, R. A. NELSON†, D. SAYRE†, P. B. SHERIDAN†,
H. STERN†, I. ZILLER†, R. A. HUGHES§, AND R. NUTT‖

Introduction

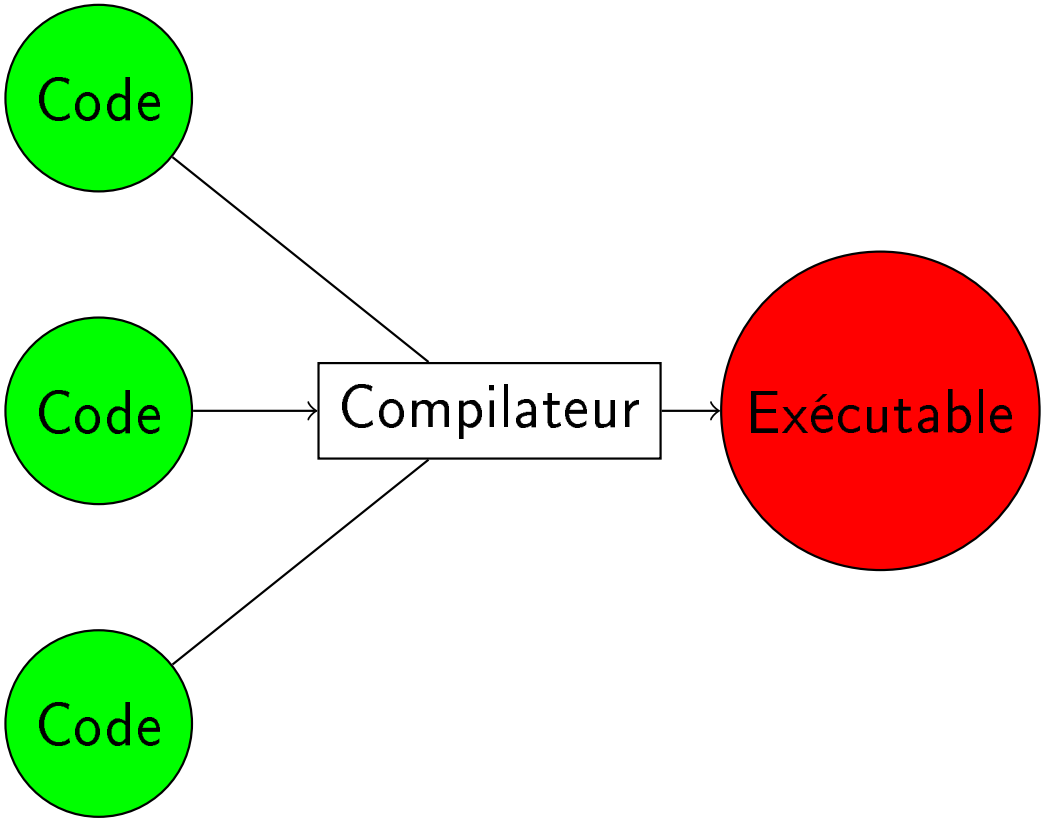
THE FORTRAN project was begun in the summer of 1954. Its purpose was to reduce by a large factor the task of preparing scientific problems for IBM's next large computer, the 704. If it were possible for the 704 to code problems for itself and produce as good programs as human coders (but without the errors), it was clear that large benefits could be achieved. For it was known that about two-thirds of the cost of solving most scientific and engineering problems on large computers was that of problem preparation. Furthermore, more than 90 per cent of the elapsed time for a problem was usually devoted to planning, writing, and debugging the program. In many cases the development of a general plan for solving a problem was a small job in comparison to the task of devising and coding machine procedures to carry out the plan. The goal of the FORTRAN project was to enable the programmer to specify a numerical procedure using a concise language like that of mathematics and obtain automatically from this specification an efficient 704 program to carry out the procedure. It was expected that such a system would reduce the coding and debugging task to less than one-fifth of the job it had been.

Two and one-half years and 18 man years have elapsed since the beginning of the project. The FORTRAN system is now complete. It has two components: the FORTRAN language, in which programs are written, and the translator or executive routine for the 704 which effects the translation of FORTRAN language programs into 704 programs. Descriptions of the FORTRAN language and the translator form the principal sections of this paper.

The experience of the FORTRAN group in using the system has confirmed the original expectations concerning reduction of the task of problem preparation and the efficiency of output programs. A brief case history of one job done with a system seldom gives a good measure of its usefulness, particularly when the selection is made by the authors of the system. Nevertheless, here are the facts about a rather simple but sizable job. The programmer attended a one-day course on FORTRAN and spent some more time referring to the manual. He then programmed the job in four hours, using 47 FORTRAN statements. These were compiled by the 704 in six minutes, producing about 1000 instructions. He ran the program and found the output incorrect. He studied the output (no tracing or memory dumps were used) and was able to localize his error in a FORTRAN statement he had written. He rewrote the offending statement, recompiled, and found that the resulting program was correct. He estimated that it might have taken three days to code this job by hand, plus an unknown time to debug it, and that no appreciable increase in speed of execution would have been achieved thereby.

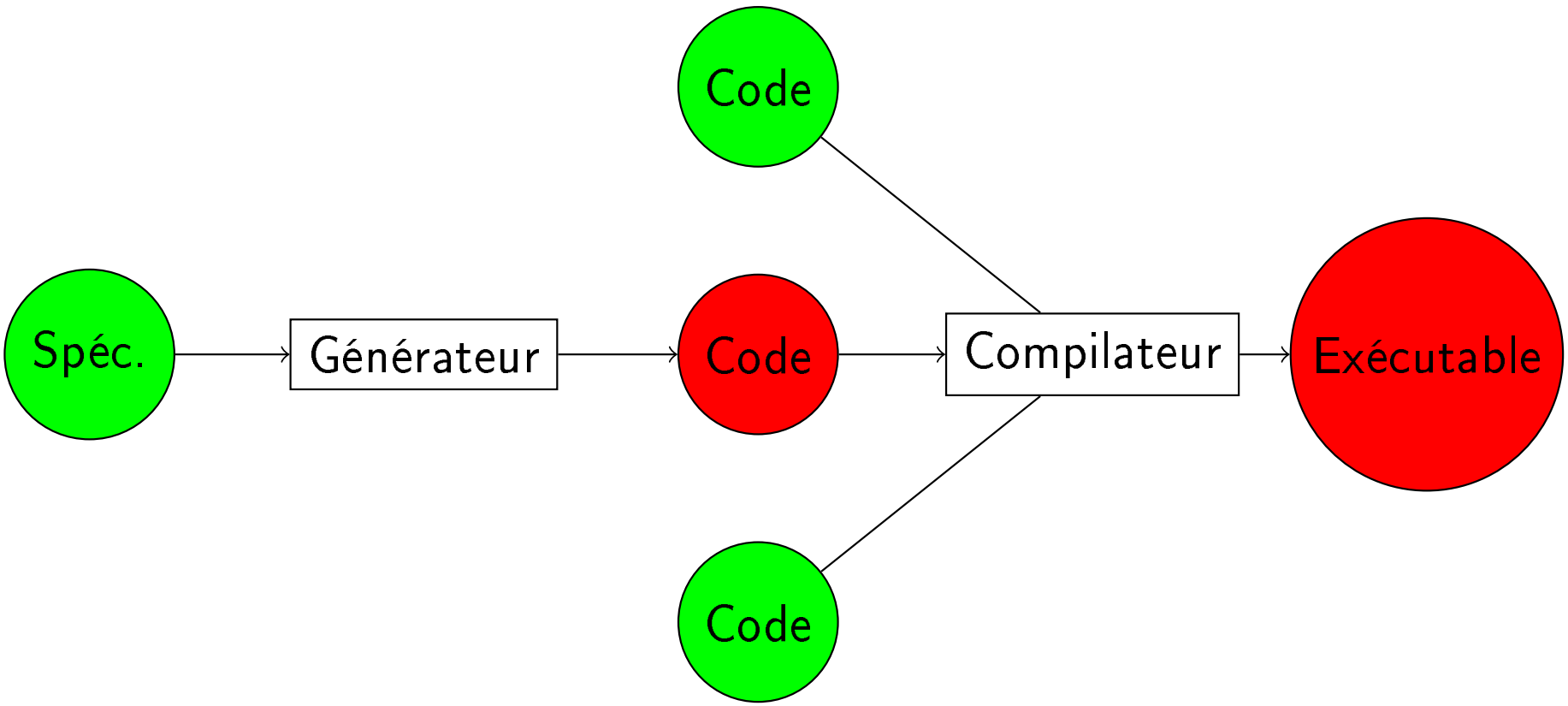
† Internat'l Business Machines Corp., New York, N. Y.
‡ Mass. Inst. Tech., Computation Lab., Cambridge, Mass.
§ Radiation Lab., Univ. of California, Livermore, Calif.
‖ United Aircraft Corp., East Hartford, Conn.

Écrire du code et compiler



- vert : écrit par le programmeur
- rouge : engendré

Utilisation d'un générateur



- vert : écrit par le programmeur
- rouge : engendré

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).

Différents générateurs

- Existent pour presque tous les langages de programmation.
- Le premier générateur était *lex*, publié en 1975 par Mike Lesk et Eric Schmidt. Engendre du code en C.
- Successeur : *flex*, 1987.
- Les générateurs modernes sont souvent issus de flex. Nous utilisons ici un générateur pour OCaml : *ocamllex*.
- Les générateurs pour des autres langages de programmation sont très similaires.

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 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 le flot d'entrée (un type abstrait)
- La fonction va
  - lire du flot d'entrée le mot le plus long décrit par une des expressions rationnelles.
  - évaluer l'expression *action* associée, et renvoyer sa valeur
  - s'il y a plusieurs possibilités pour le mot le plus long : on prend la première.

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'érire un 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 *)

(* fontion 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 :
  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

Un exemple avec nomage d'une partie du mot

- Exemple (un peu artificiel) : des entiers qui peuvent être entre une séquence de symboles `[` et une séquence de `]`
- On ne peut pas, *avec les expressions rationnelles*. assurer que le nombre de `[` est égal au nombre de `]`. (lemme d'itération !)
- Il est facile d'écrire la bonne expression rationnelle, mais on veut aussi extraire du lexeme trouvé la partie consistant de chiffres, pour pouvoir la convertir en valeur entière.

Le fichier numbers.mll

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

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

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
- 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 ⇒ 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 ⇒ token BEGIN.
- Troisième appel à `next_token` : les règles (1) et (5) s'appliquent mais la dernière reconnaît un lexeme plus long ⇒ token ID.

Comment reconnaître les mots clefs sans catégories dédiées ?

- Problème : avec une règle par mot clef l'automate peut devenir trop grand
- Message : *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 I

```
{
  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://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html`
  Seulement les deux premières sections 15.1 et 15.2 sont pertinentes car nous n'allons pas utiliser `ocamlyacc` (nous avons mieux !)
- Documentation du module `Lexing` :
  `https://caml.inria.fr/pub/docs/manual-ocaml/libref/Lexing.html`