Langages et Automates: Lex et Yacc

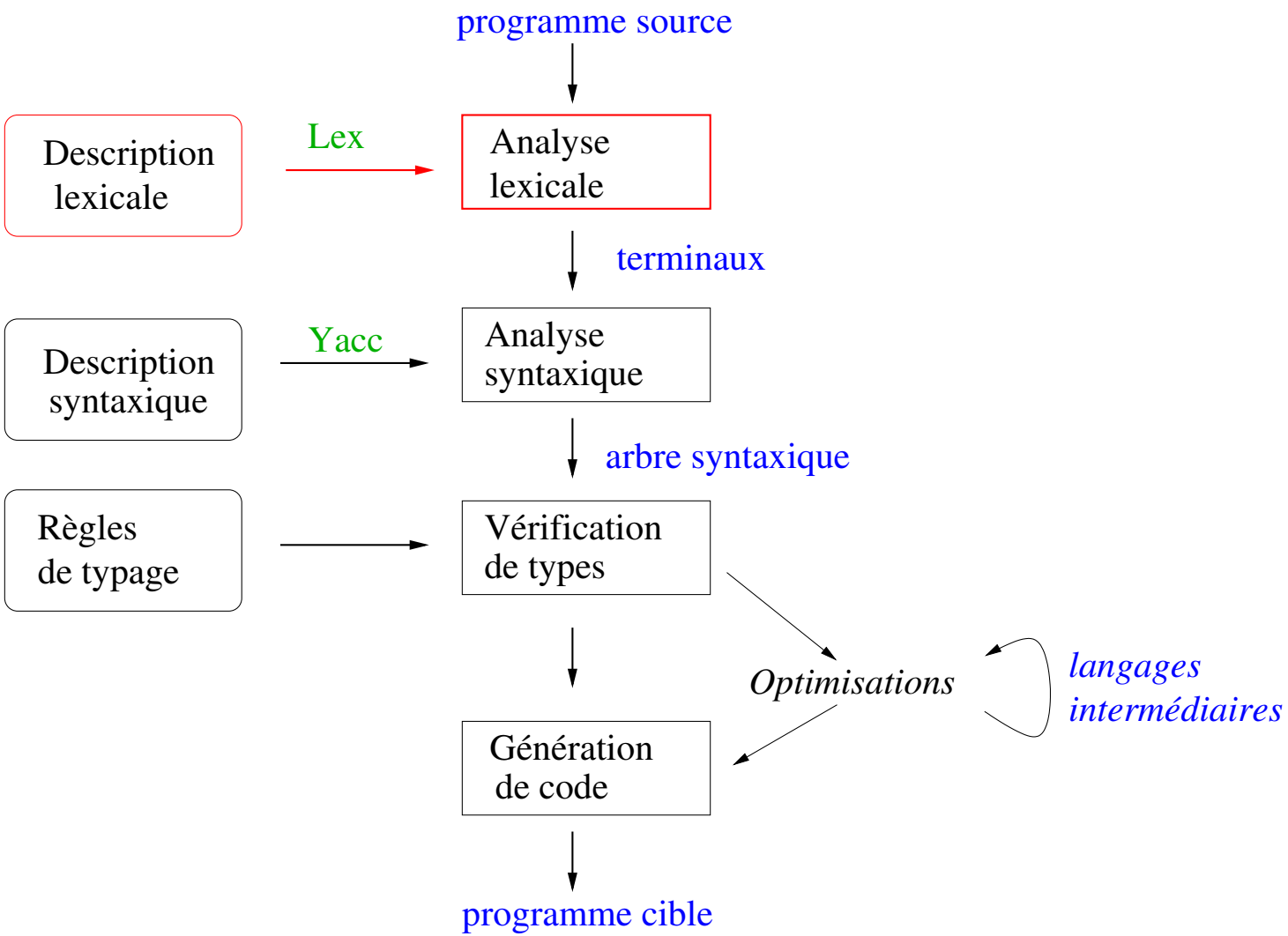
Martin Strecker

INU Champollion

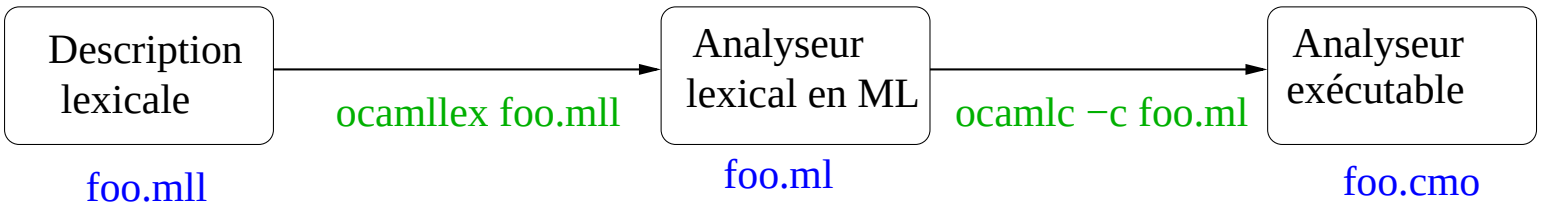
Année 2021/2022

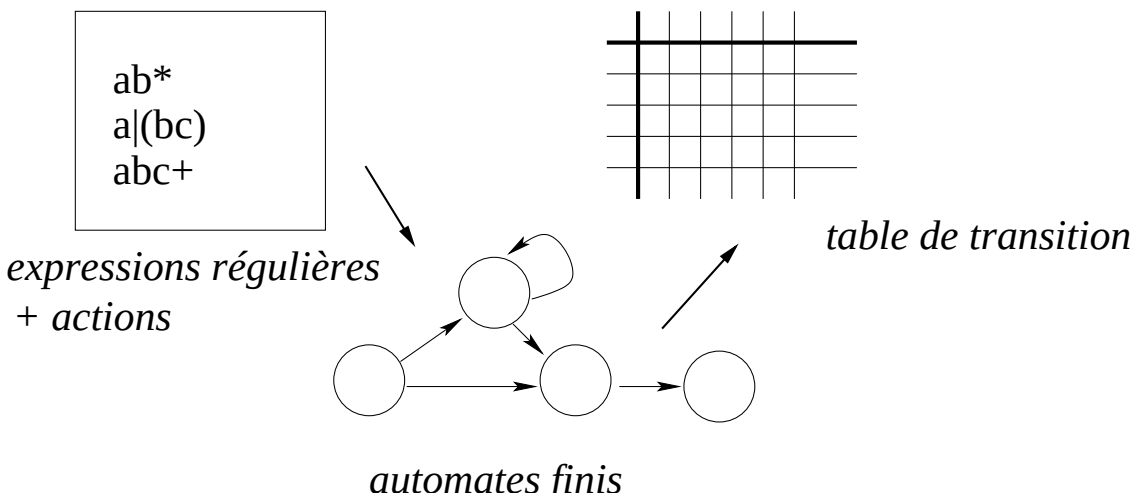
Plan

Situation dans le processus de compilation



Fonctionnement de Lex





Syntaxe des expressions régulières (1)

Caractères simples

`'x'`	le caractère `x`
`_`	(souligné) tout caractère
`'\n'`	newline

Classes de caractères

`['x''y''z']`	l'un des caractères `x`, `y`, `z`
`['A'-'Z']`	les car. `A...Z`
`[^'x''y''z']`	tout caractère sauf `x`, `y`, `z`
`[^'A'-'Z']`	tout caractère sauf `A...Z`
`"abc"`	la chaîne de caractère `abc`
`eof`	fin de l'entrée

Syntaxe des expressions régulières (2)

Opérateurs

`rs`	concaténation
`r|s`	alternative
`r?`	élément optionnel
`r*`, `r+`	répétition (0 fois ou plus, 1 fois ou plus)

. . . et beaucoup plus.

Détails : manuel d'Ocaml

Organisation de la description lexicale (1)

```
Déclarations / définitions pour le programme ML
{
  exception Lexerror
}
Abréviations d'expressions régulières
let id = ['a'-'z''A'-'Z']['a'-'z''A'-'Z''0'-'9']*
Expressions régulières et actions associées
rule main = parse
    id ";"        { print_string "..." }
  | ['0'-'9']+    { print_string "..." }
Autres fonctions et programme principal
{
main (Lexing.from_channel stdin)
}
```

Organisation de la description lexicale (2)

Analyse d'une chaîne de caractères

- entrée : un *lex buffer*
  Ici : `(Lexing.from_channel stdin)`
- analyse : fonction appliquée au *lex buffer*
  Ici : `main`
  En général : Toute fonction `f` définie par
  `rule f` a_1 `...` a_n `= parse ...`
- Sortie : élément d'un type de données. Ici : `unit`

Librairie `Lexing` :

`lexbuf`	variable prédéfinie dans la directive `parse`
`from_channel`	crée un *lex buffer*
	Ex. : `Lexing.from_channel stdin`
`lexeme`	dernier mot reconnu
	Ex. : `Lexing.lexeme lexbuf`

Détails : manuel d'Ocaml

Lex : Exemple

```
(* parameterized rule *)
rule count nl nc = parse
      (* lexbuf becomes  explicit argument *)
    '\n'   { count (nl + 1) (nc + 1) lexbuf }
  | _      { count nl (nc + 1) lexbuf }
  | eof    { (nl, nc) }

{
  let lexbuf = Lexing.from_channel stdin in
  let (nl, nc) = count 0 0 lexbuf in
    Format.printf
     "nb␣of␣lines␣=␣%d,␣nb␣of␣chars␣=␣%d\n" nl nc
}
```

Lex : Comment compiler ?

. . . à la main :

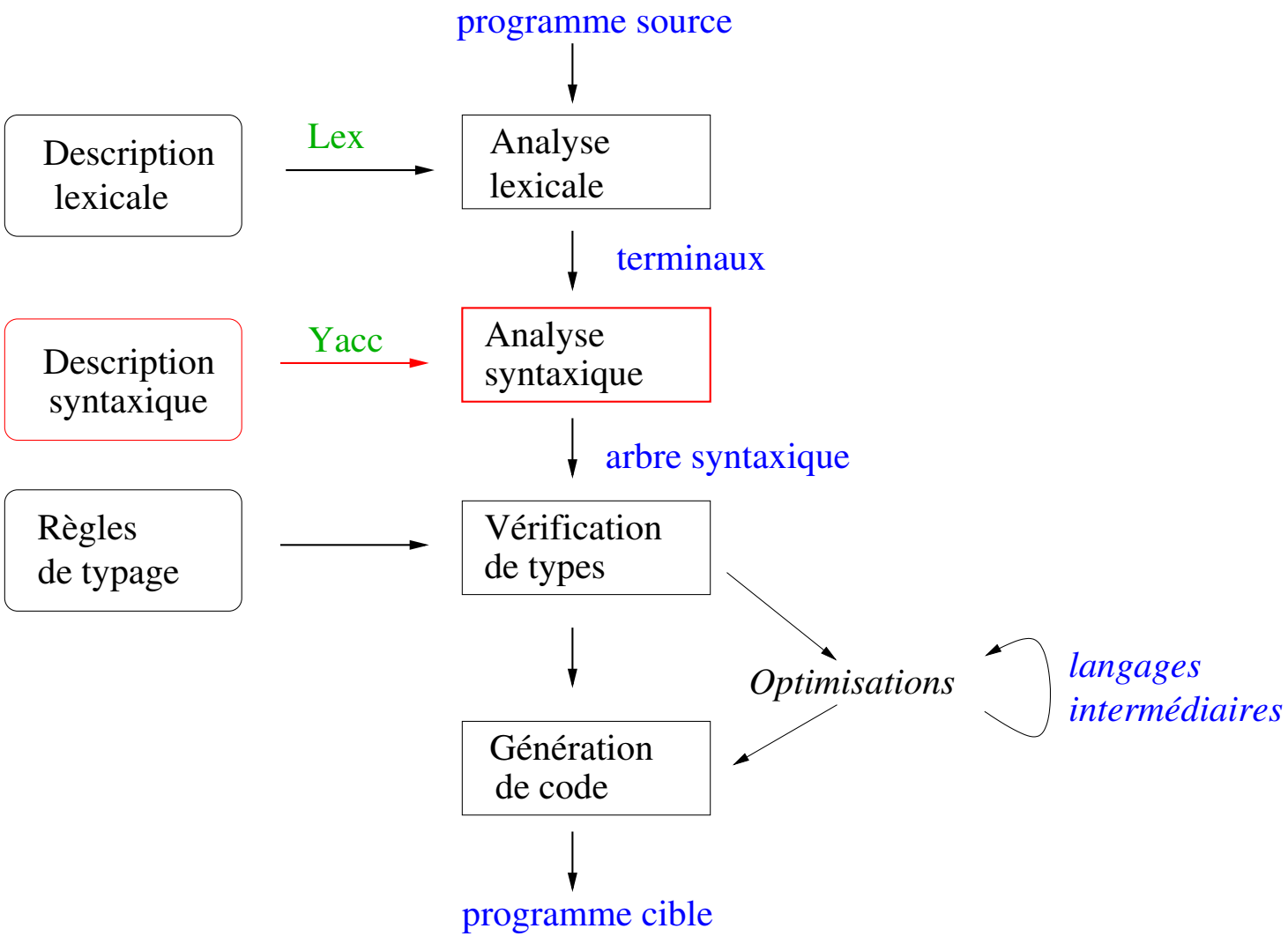
- `ocamllex linecount.mll` $\rightsquigarrow$ `linecount.ml`
- `ocamlc -o linecount linecount.ml` $\rightsquigarrow$ `linecount`
- Exécuter `linecount < fichier`

. . . éléments d'un Makefile :

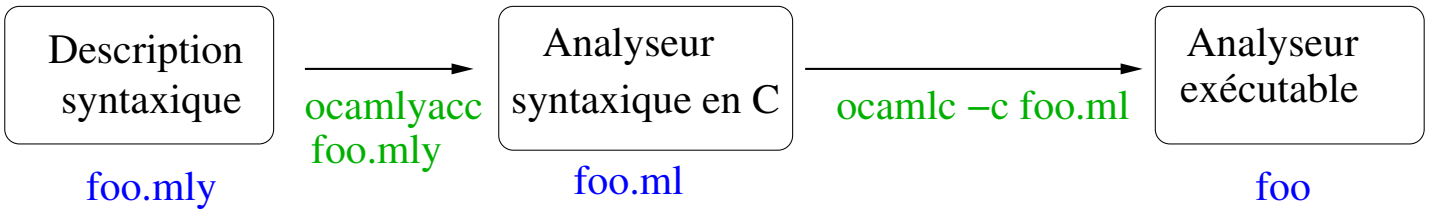
```
linecount: linecount.cmo
	ocamlc -o linecount $<
linecount.cmo: linecount.ml
	ocamlc -c $<
linecount.ml: linecount.mll
	ocamllex $<
```

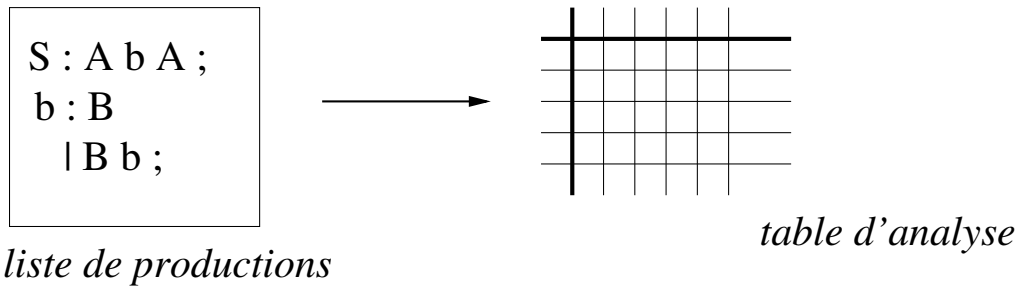
Plan

Situation dans le processus de compilation



Fonctionnement de Yacc





Organisation de la description syntaxique

```
Déclarations / définitions pour le programme ML
%{  open Lang   (* partie optionnelle *)
%}
Déclaration de propriétés de symboles
%start s
%%
Règles de production et actions sémantiques
s : A b A       { printf "..." }
;
b : B           { printf "..." }
  | B b         { printf "..." }
;
%%
Fonctions et programme principal
let main  = ...   (* partie optionnelle *)
```

Déclaration de propriétés de symboles

Racine (déclaration obligatoire)

```
%start non-terminal
```

Terminaux

```
%token<type> liste de terminaux
```

Non-terminaux (décl. obligatoire pour racine)

```
%type<type> liste de non-terminaux
```

Associativité et priorité des terminaux

```
%left liste de terminaux
%right liste de terminaux
```

Exemple :

```
%left PLUS MINUS
%left MULT DIV
```

Actions sémantiques

En général : Expr. d'OCAML comprises entre `{ ... }`

```
b : B          { print_string "règle b1" }
  | B b        { print_string "règle b2" }
;
```

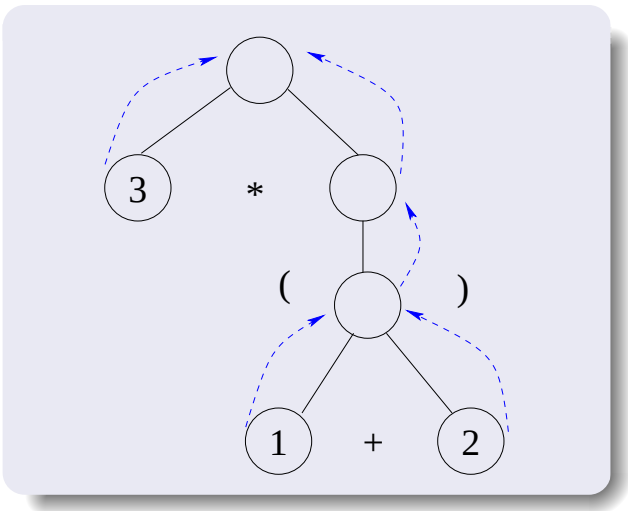
Actions sémantiques

En général : Expr. d'OCAML comprises entre `{ ... }`

```
b : B          { print_string "règle b1" }
  | B b        { print_string "règle b2" }
;
```

Accès aux sous-arbres :

```
e : e PLUS e      { $1 + $3 }
  |  ....
  | LPAR e RPAR     { $2 }
;
```



Conflits

Grammaire ambiguë :

```
s : A b C     {}
|   A B C     {}
;
b : B         {}
;
```

Analyse de conflits :

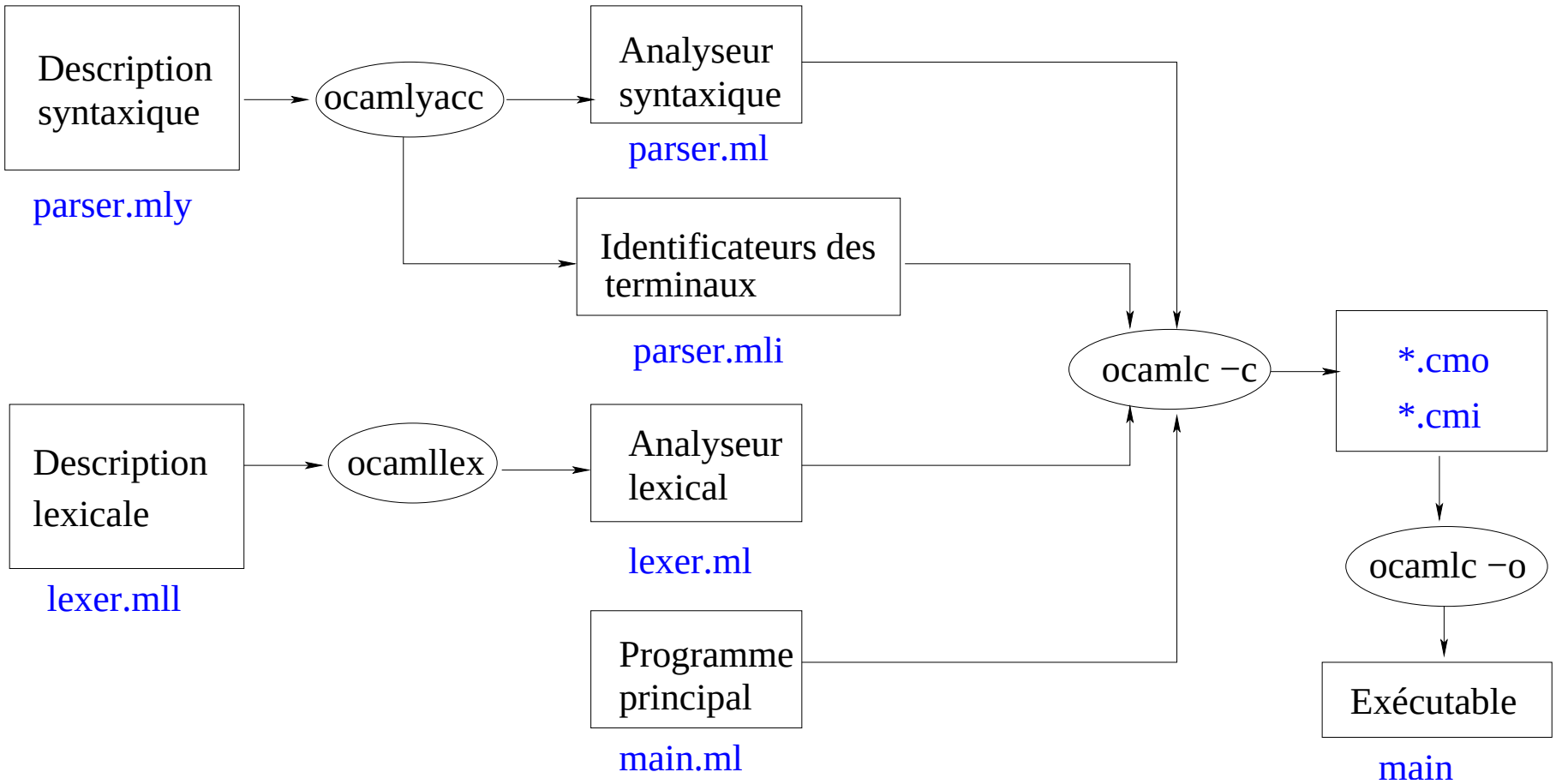
`ocamlyacc -v foo.mly` crée `foo.output`

```
5: shift/reduce conflict (shift 7, reduce 3) on C
state 5
        s : A B . C   (2)
        b : B .   (3)

        C  shift 7
```

Plan

Schéma de compilation



Exemple : Fichier Lex

```
{ open Parser
  exception Eof   ... }

let blancs  = [' ''\t']+
let boolean = 'T'|'F'    ...

rule token = parse
  blancs           { token lexbuf } (* appel recursif *)
| boolean as b     { if b = 'T' then (BCONST true)
                                else (BCONST false) }
| num as i         { INTCONST (int_of_string i)}
| '^'              { OPERATOR }
| '\n'             { EOL }
| eof              { raise Eof }
| _                { Printf.printf
          "unrecogized␣'%s'\n" (Lexing.lexeme lexbuf);
                     raise Lexerror }
```

Exemple : Fichier Yacc (1)

Déclarations :

```
%token <bool> BCONST
%token <int> INTCONST
%token OPERATOR
%token EOL
%left  OPERATOR
%type <bool> term expr formule
%start formule
```

Exemple : Fichier Yacc (2)

Grammaire :

```
%%
formule: expr EOL { $1 }
;
expr: expr OPERATOR expr { $1 && $3 }
 | term { $1 }
;
term: BCONST { $1 }
 | INTCONST { not ($1 = 0) }
;
```

voir `parser.mly`

Exemple : Programme principal

```
let _ =
  try
   let lexbuf = Lexing.from_channel stdin in
    while true do
      let result = Parser.formule Lexer.token lexbuf in
      print_string ("Result: " ^ (string_of_bool result));
      print_newline(); flush stdout
    done
  with Lexer.Eof ->
    Format.printf "Finished\n"; exit 0
```

voir `binexpr.ml`