IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

# TP2&3 – Compilation

## L'analyse lexicale et syntaxique avec Ocamllex et Menhir

### Ingénierie du développement logiciel – F2B304

## 1 Introduction

La compilation repose en premier lieu sur la reconnaissance de programmes dans un flux de caractères. Le but de ce TP est de vous aider à découvrir les phases dites d'*analyse lexicale* et d'*analyse syntaxique*. L'analyse lexicale consiste à reconnaître dans des suites de caractères des mots du langage. L'analyse syntaxique consiste ensuite à regrouper ces mots pour reconnaître des phrases. Dans le vocabulaire de la compilation, on parlera de *lexèmes* ou *unités lexicales* pour les mots et d'*unités syntaxiques* pour les phrases. Il convient également de remarquer que pour les besoins des phases suivantes de compilation, on représente les unités syntaxique sous forme d'arbre : l'*arbre syntaxique abstrait (AST)*. La figure 1 représente cet enchainement. Il convient de remarquer qu'en général, l'analyseur lexical est piloté par l'analyseur syntaxique qui demande les unités lexicales suivant les besoins qu'il a.

En règle générale, les analyseurs sont des automates efficaces qui recherche les motifs lexicaux et syntaxiques en utilisant des expressions régulières. Comme programmer un automate est difficile des langages spécifiques à ces deux domaines ont été créés.

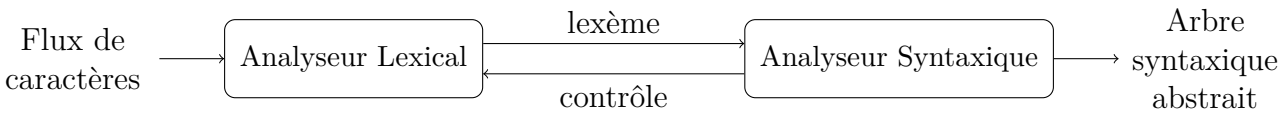



FIGURE 1 – L'analyse lexicale et syntaxique.

# 2 Les outils

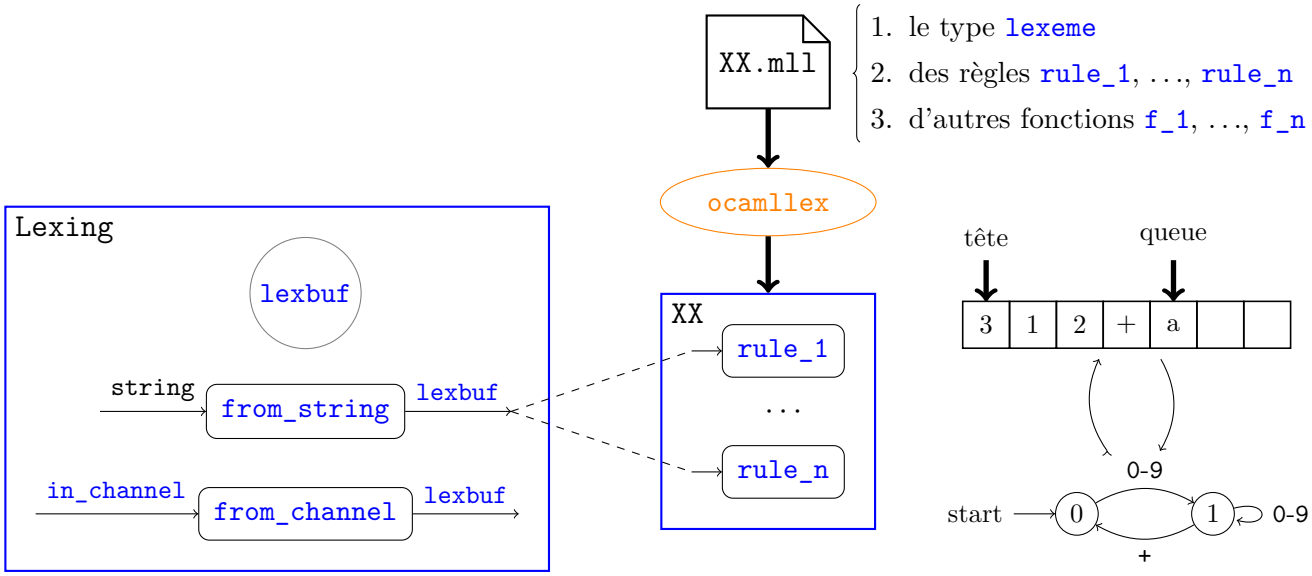
Durant ce TP, nous allons découvrir les outils `ocamllex` et `menhir`[1], versions Ocaml des outils Unix standard `lex` et `yacc`. Le manuel d'ocaml contient une description d'`ocamllex` `http://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html` et la page de menhir contient sa documentation `http://gallium.inria.fr/~fpottier/menhir/menhir.html.fr`. Le chapitre 16 de *Real World Ocaml* pourra vous aider dans cette partie de l'UV.

`ocamllex` permet la construction d'analyseurs lexicaux sous la forme d'automates déterministes. Pour cela, il utilise un langage dédié qui permet de spécifier par des règles les actions à exécuter lors de la reconnaissance de chaînes. À partir de ces règles, il produit un programme Ocaml. Il est, en général, utilisé pour transformer une chaîne de caractères en une suite de lexèmes.

`menhir` permet la construction d'analyseurs syntaxiques sous la forme d'automates déterministes. Pour cela, il utilise un langage dédié qui permet de spécifier par des règles les actions à exécuter lors de la reconnaissance de phrases. À partir de ces règles, il produit un programme Ocaml. Il est, en général, utilisé pour transformer une suite de lexèmes en un arbre syntaxique.

L'outil `menhir` produit un automate à pile LR(1). Pour utiliser `menhir`, il faudra l'installer `opam install menhir`.

# 3 Utilisation d'`ocamllex`




## 3.1 La syntaxe des fichiers `ocamllex`

Les fichiers `ocamllex` portent l'extension `.mll` et suivent la syntaxe suivante :

```
{ (* du code Ocaml *) }
let ident = regexp
```

1. Un outil plus ancien `ocamlyacc` existe aussi qui colle un peu plus à `yacc`. Menhir accepte les fichiers au format `ocamlyacc` et fournit des messages d'erreurs plus faciles à comprendre.

| Expression | Signification |
|---|---|
| ' *char* ' | le caractère *char* |
| _ | n'importe quel caractère |
| *eof* | la fin du tampon |
| " *string* " | la chaîne de caractères *string* |
| [ *ens* ] | n'importe quel caractère appartenant à l'ensemble *ens*. Cet ensemble peut être un singleton 'c', une énumération 'c1'-'c2' (tous les caractères entre c1 et c2) ou une union de plusieurs ensembles |
| [^ *ens* ] | n'importe quel caractère n'appartenant pas à l'ensemble *ens* |
| *regexp* * | la concaténation de zéro ou plusieurs chaînes correspondant à *regexp* |
| *regexp* + | la concaténation de une ou plusieurs chaînes correspondant à *regexp* |
| *regexp* ? | soit la chaîne vide soit les chaînes correspondant à *regexp* |
| *regexp1* \| *regexp2* | toutes les chaînes correspondant à *regexp1* ou *regexp2* |
| *regexp1* *regexp2* | toutes les concaténations de deux chaînes, la première correspondant à *regexp1* et la seconde à *regexp2* |
| (*regexp*) | les mêmes chaînes que l'expression régulière *regexp* |
| *ident* | l'expression régulière liée précédemment à *ident* par une déclaration |
| *regexp* as *ident* | lie le résultat de l'application de l'expression régulière à la variable *ident* |

FIGURE 2 – La syntaxe des expressions régulières d'`ocamllex`

```
let ident = regexp
rule ident [ident1 ... identn] = parse
 regexp { (* du code Ocaml *) }
| regexp { (* du code Ocaml *) }
and ident [ident1 ... identn] = parse
 regexp { (* du code Ocaml *) }
| regexp { (* du code Ocaml *) }
{ (* du code Ocaml *) }
```

Les deux sections comportant du code Ocaml en début et en fin de fichier sont facultatives. Elles contiennent du code définissant des éléments (types, fonctions, etc) nécessaires au traitement. La dernière section peut définir des fonctions utilisant les règles d'analyse d'ensemble lexicaux données dans la section médiane.

La série de déclarations `let` précédant la définition des règles permet de nommer certaines expressions régulières. Ce nom peut alors être utilisé dans la définition des règles. Les expressions régulières d'`ocamllex` suivent la syntaxe présentée figure 2.

Chaque règle (`rule`) produit une fonction de même nom (il doit donc être un identifiant ocaml). Si la règle possède des arguments, la fonction obtenue aura ces arguments en plus d'un dernier argument (ajouté) qui est un tampon (*buffer*) de type `Lexing.lexbuf`. Cette fonction recherche alors les expressions régulières de la liste correspondante qui représentent un préfixe du buffer. L'expression régulière correspondant au plus grand préfixe possible est sélectionnée [2] et l'action associée est effectuée. Le module `Lexing` contient quelques fonctions de manipulation des tampons lexicaux que le programmeur peut utiliser pour définir ses traitements. Ce module contient, entre autre [3] :

2. S'il existe deux expressions régulières reconnaissant des chaînes de même taille, la première dans l'ordre de définition du fichier est utilisée.

3. Pour plus de détails : `http://caml.inria.fr/pub/docs/manual-ocaml/libref/Lexing.html`.

- le type `lexbuf` ;
- deux constructeurs pour ce type : `from_channel` et `from_string` qui respectivement crée un tampon à partir d'un canal d'entrée sortie ou d'une chaîne de caractères ;
- les fonctions :
  - `lexeme` : qui renvoie la chaîne reconnue par l'expression régulière. En général, il est plus simple d'utiliser la construction syntaxique `as` qui permet en plus d'extraire une sous-partie de la chaîne reconnue.
  - `lexeme_start` : qui renvoie l'index de position du début de la portion de chaîne reconnue.
  - `lexeme_end` : qui renvoie l'index de position de fin de la portion de chaîne reconnue.

Chaque expression régulière est compilée par l'outil en automate. L'ensemble des automates sont fusionnés en un seul automate. Cet automate est alors determinisé puis minimisé. Son code est inséré entre les deux portions de code Ocaml de début et de fin du fichier `.mll` pour former un fichier `.ml` associé (qui réalise l'analyse lexical).

## 3.2 Un premier exemple

Commençons par examiner l'exemple ci-dessous, le fichier `lexformule-0.mll`. Il s'agit de reconnaître des expressions booléennes simples. Le prélude est utilisé pour définir le type des lexèmes que nous souhaitons reconnaitre, appelé ici `lexeme`. Ce type décrit les quatres éléments reconnus : les opérateurs `AND` et `OR` ainsi que les constantes `TRUE` et `FALSE`. Il définit également un élément supplémentaire pour signaler la fin du buffer `EOF`. Ensuite, on définit une expression régulière pour reconnaître les espaces. Enfin, on définit une règle `nexttoken` qui va parcourir le buffer pour en extraire le prochain lexème. Tant qu'il y a des espaces, on va chercher plus loin, en fin de fichier on renvoie une fin de fichier et sinon pour chacune des formes de chaînes de caractères qui nous intéresse, on renvoie l'élément correspondant du type `lexeme`.

```
{
 type lexeme =
   | EOF
   | AND
   | OR
   | TRUE
   | FALSE
}
let space = [' ' '\t' '\n']
rule nexttoken = parse
   space+ { nexttoken lexbuf }
 | eof    { EOF }
 | "et"   { AND }
 | "ou"   { OR }
 | "vrai" { TRUE }
 | "faux" { FALSE }
```

Ce fichier[4] peut être compilé par `ocamllex`, il produit alors un fichier `lexformule-0.ml`. Dans ce fichier est alors définie la fonction `nexttoken` de type `Lexing.lexbuf -> lexeme` qui réalise l'automate correspondant à la règle. On peut alors tester cette fonction dans l'interpréteur :

4. Les sources sont sur le subversion.

```
utop # #use "lexformule-0.ml";;
type lexeme = EOF | AND | OR | TRUE | FALSE
val nexttoken : Lexing.lexbuf -> lexeme = <fun>
utop # let buffer = Lexing.from_string "vrai et faux";;
val buffer : Lexing.lexbuf =
  {Lexing.refill_buff = <fun>; lex_buffer = "vrai et faux"; lex_buffer_len = 12;
   lex_abs_pos = 0; lex_start_pos = 0; lex_curr_pos = 0; lex_last_pos = 0;
   lex_last_action = 0; lex_eof_reached = true; lex_mem = [||];
   lex_start_p = {Lexing.pos_fname = ""; pos_lnum = 1; pos_bol = 0; pos_cnum = 0};
   lex_curr_p = {Lexing.pos_fname = ""; pos_lnum = 1; pos_bol = 0; pos_cnum = 0}}
utop # nexttoken buffer;;
- : lexeme = TRUE
utop # nexttoken buffer;;
- : lexeme = AND
utop # nexttoken buffer;;
- : lexeme = FALSE
utop # nexttoken buffer;;
- : lexeme = EOF
```

## Exercice 1 (*Des variables dans les formules*)

▷ **À partir du code ci-dessus, réaliser un analyseur lexical qui va également reconnaître des variables. Une variable doit commencer par une lettre et être suivi par un nombre quelconque de lettres, chiffres ou du caractère `_`.**

Pour extraire le nom de la variable reconnu, pensez à utiliser la construction `as`.

### 3.3 Des précisions

Une règle `ocamllex` peut être considérée comme une fonction car `ocamllex` génère une fonction de même nom. Cette fonction est récursive comme on l'a vu dans l'exemple précédent (`nexttoken` appelle `nexttoken` lors de l'élimination des espaces).

On peut également lui ajouter des paramètres. Nous allons l'illustrer avec un petit exemple qui ne nécessiterait sans doute pas l'utilisation d'`ocamllex`. Il s'agit de réaliser un petit automate qui va compter les occurrences du caractère `'a'` dans une chaîne de caractères. Pour cela, nous définissons une règle qui prend en paramètre (`value`) le nombre de `'a'` déjà rencontrés. Lorsque l'automate rencontre un `'a'`, il s'appelle récursivement avec une valeur incrémentée de 1. Si la chaîne est terminée, il retourne le nombre de `'a'` et sinon, il se rappelle sans modifier le nombre de `'a'`. La fonction voulue est alors définie dans le postlude.

```
{ }
let not_a = [^'a']*
rule count value = parse
  | not_a  { count value lexbuf }
  | 'a'    { count (value + 1) lexbuf }
  | eof    { value }
{
 let compte_a s =
```

```
  let buffer = Lexing.from_string s in
   count 0 buffer
}
```

Ce fichier, une fois compilé par `ocamllex`, et chargé dans l'interpréteur peut être testé.

```
utop # #use "compte_a.ml";;
val count : int -> Lexing.lexbuf -> int = <fun>
val compte_a : string -> int = <fun>
utop # compte_a "eratatata";;
- : int = 4
```

# 4 Des expressions

Cette section revient sur l'évaluateur d'expression arithmétiques vu dans le TP d'introduction à ocaml. Nous allons le doter d'un analyseur lexical qui sera ensuite complèté d'un analyseur syntaxique.

## Exercice 2 (*Expression correcte*)

▷ **Question 2.1 :**
**Construire un analyseur lexical permettant de reconnaître des expressions arithmétiques basiques composées de réels, des quatre opérateurs de base et de variables.**

Le code Ocaml ci-dessous vous permet de lire un fichier dont le nom est passé en paramètre de la fonction `compile` :

```
let compile file =
 print_string ("File "^file^" is being treated!\n");
 try
  let input_file = open_in file in
  let lexbuf = Lexing.from_channel input_file in
  (* Travail sur le fichier *)
  close_in (input_file)
 with Sys_error s ->
  print_endline ("Can't find file '" ^ file ^ "'")
```

▷ **Question 2.2 :**
**À partir du code précédent, étendez votre programme pour qu'il lise l'expression dans un fichier et affiche tous les lexèmes du fichier.**

## Exercice 3 (*Gestion des erreurs*)

Il s'agit d'étendre le code précédent pour fournir des messages d'erreur plus parlant. En l'occurrence, il s'agit d'indiquer en quel point du fichier l'erreur (lexicale) se produit. Pour cela, nous allons utiliser le type `position` défini dans le module `Lexing`. Ce type permet de mémoriser et de manipuler sous forme structurée la position dans un fichier :

```
type position = {
 pos_fname : string; (* nom du fichier *)
 pos_lnum : int;    (* numero de la ligne *)
 pos_bol : int;     (* nb de caracteres entre le debut du fichier et celui de la ligne *)
 pos_cnum : int;    (* nb de caracteres depuis le debut du fichier *)
}
```

Par défaut, l'analyseur lexical généré ne met à jour que le dernier champs (`pos_cnum`). C'est donc aux actions de gérer les autres champs.

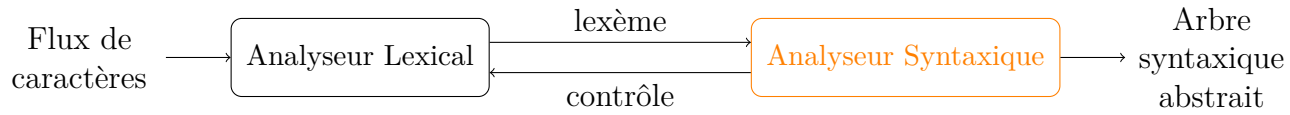
Les fonctions `lexeme_start_p` et `lexeme_end_p` du module `Lexing` permettent de récupérer respectivement la position de début et de fin de l'unité lexicale en cours d'analyse.

▷ **Modifier votre code pour atteindre l'objectif fixé.**

Vous pourrez également utiliser `ocamlbuild` pour produire un exécutable directement. Il convient alors d'ajouter le ligne suivante à votre fichier pour déclenche la fonction `compile` en lui passant en paramètre le contenu de la ligne de commande.

```
let _ = Arg.parse [] compile ""
```

# 5 Utilisation de `menhir`




Les fichiers `menhir` portent l'extension `.mly` et suivent la syntaxe suivante :

```
%{
 (* du code Ocaml *)
%}
 /* declarations des symboles */
%%
 /* regles */
%%
 (* du code Ocaml *)
```

Les deux sections comportant du code Ocaml en début et en fin de fichier sont facultatives. Elles contiennent du code définissant des types, des fonctions, etc nécessaires au traitement. La dernière section peut définir des fonctions utilisant les règles d'analyse syntaxique données dans la section médiane.

## 5.1 Les déclarations

Les déclarations possibles sont les suivantes :

- `%token symbol ... symbol` : déclare les symboles comme des unités lexicales, c'est-à-dire, ils sont ajoutés comme constructeurs constants au type `token`.
- `%token <type> symbol ... symbol` : déclare les symboles comme des unités lexicales, c'est-à-dire, ils sont ajoutés comme constructeurs ayant comme paramètre le type spécifie au type `token`.
- `%start symbol ... symbol` : déclare les symboles comme des points d'entrée, c'est-à-dire, une fonction d'analyse est définie pour chacun des symboles. Ces symboles doivent avoir un type déclaré par la déclaration suivante.
- `%type <type> symbol ... symbol` : déclare le type des attributs sémantiques des symboles, c'est-à-dire, le résultat de la règle d'analyse de même nom. Ces types ne sont obligatoires que pour les points d'entrée.
- la précédence et l'associativité :
  - `%left symbol ... symbol` :
  - `%right symbol ... symbol` :
  - `%nonassoc symbol ... symbol` :

  L'ordre d'apparition de ces lignes fixe la précédence. Les premiers définis sont moins prioritaires que les suivants. Les symboles figurant sur la même ligne ont la même priorité.

## 5.2 Les règles

Les règles de syntaxe ont la forme suivante :

```
nonterminal :
  | symbol ... symbol { (* action semantique *) }
  | ...
  | symbol ... symbol { (* action semantique *) }
```

Une action sémantique contient du code Ocaml qui permet de donner la valeur sémantique du non terminal dans le cas correspondant. Cette action sémantique peut utiliser pour son calcul la valeur sémantique de tous les terminaux et non terminaux qui apparaissent dans la production choisie. Deux modes pour acceder à leur valeur sont disponibles : (1) en nommant ces symboles dans la production ou (2) par position `$1` pour le premier symbole et jusqu'à `$9`. La deuxième possibilité est necessaire pour la compatibilité avec `ocamlyacc` mais son usage est découragé car elle crée un fort couplage entre l'action et la production (si un symbole est déplacé, le code de l'action doit être modifié.

En général, l'action sémantique consistera à construire le nœud de l'arbre syntaxique abstrait associé à la phrase reconnue.

Pour démontrer le fonctionnement de `menhir`, nous allons examiner un exemple en détail. Il s'agit de reprendre la suite de la section 3.2. Les formules logiques sont composées de conjonctions (et) et disjonctions (ou) de variables (des identificateurs). L'analyseur syntaxique va construire une version complètement parenthésée, la conjonction étant plus prioritaire que la disjonction.

```
%token AND OR EOF TRUE FALSE
%token <string> IDENT

%start formule
%type < string > formule

%%
```

```
formule:
 | c=conjonction EOF        { c }
conjonction:
 | d=disjonction AND c=conjonction { "("^d^" et "^c^")" }
 | d=disjonction           { d }
disjonction:
 | id=ident_or_const OR d=disjonction    { "("^id^" ou "^d^")" }
 | id=ident_or_const             { id }
ident_or_const:
 | id=IDENT             { id }
 | TRUE                { "true" }
 | FALSE               { "false" }
%%
```

À partir de ce fichier, la commande `menhir parseformule.mly` génère un module composé d'un fichier signature `parseformule.mli` et d'un fichier de réalisation `parseformule.ml`. Ce module généré contient :

— la déclaration du type `token` qui décrit les lexèmes :

```
type token =
  | TRUE
  | OR
  | IDENT of (string)
  | FALSE
  | EOF
```

— une fonction `formule` de type `(Lexing.lexbuf -> token) -> Lexing.lexbuf -> string` qui prend en arguments une fonction d'analyse lexicale (généralement construit avec `ocamllex`) et un tampon et renvoie la formule correspondante.

La fonction d'analyse lexicale correspond à la solution de l'exercice 1 sans la définition du type, faite maintenant dans le fichier de l'analyseur syntaxique.

**Remarque :**

*Le fichier de signature n'inclut pas l'entête du fichier* ***menhir****, donc, tous les types qui y figurent doivent être complètement qualifiés (c'est-à-dire précédé de leur module). Par conséquent, les types des points d'entrée et les types paramètres des lexèmes dans le fichier* ***menhir*** *doivent être complètement qualifiés.*

# 6 Un exercice

## Exercice 4 (*Analyse syntaxique des expressions*)

Réaliser l'analyseur syntaxique des expressions arithmétiques et connecter le résultat à l'évaluateur fait dans le TP Ocaml.