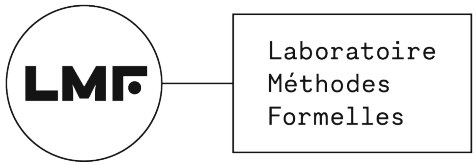


Introduction à la compilation

Polytech'Paris-Saclay– 4ème année –

Menhir : un générateur d'analyseur syntaxique

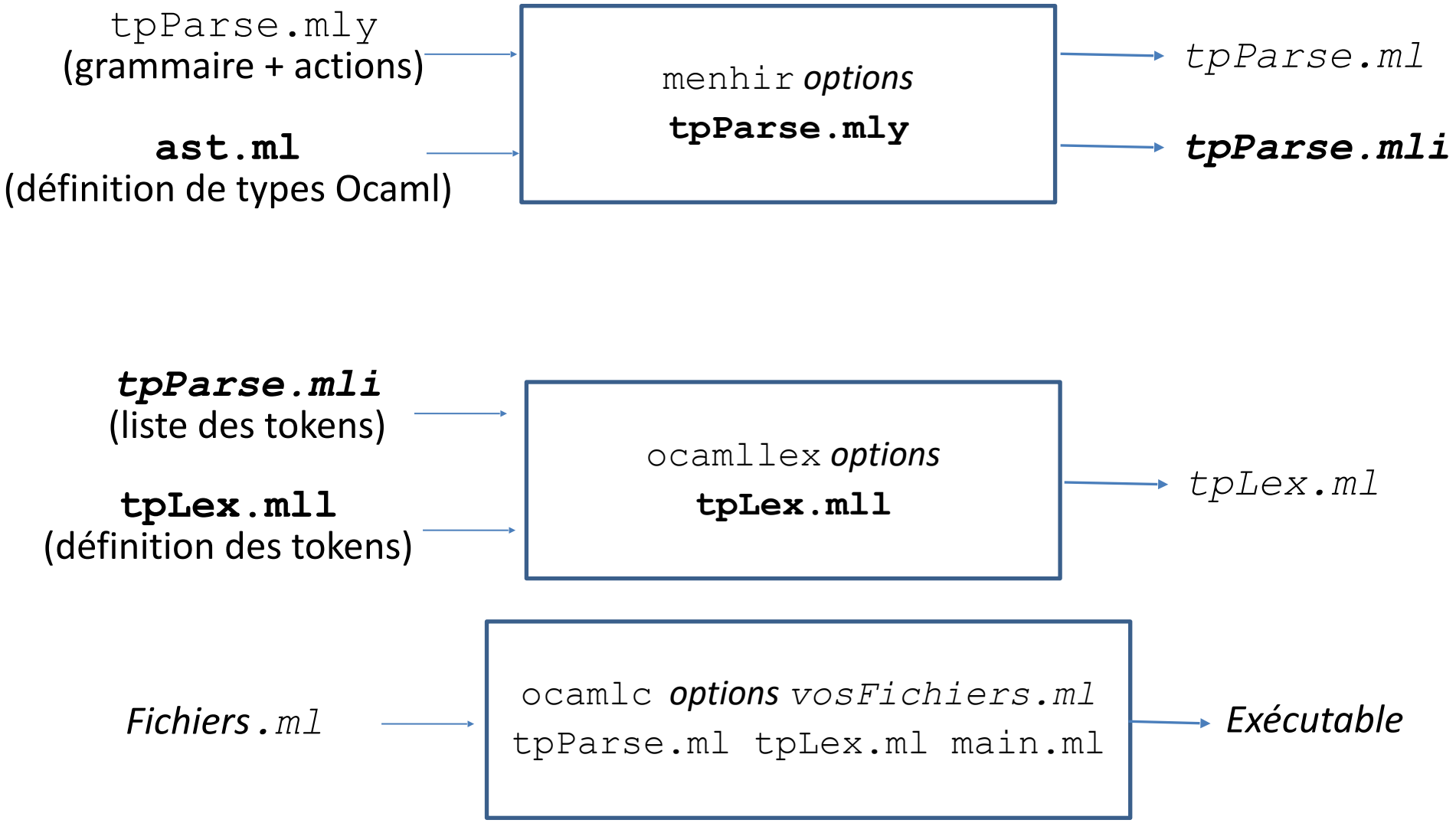
Burkhart Wolff (& Frederic Voisin)

Menhir : un générateur d'analyseur syntaxique

- Semblable à d'autres générateurs d'analyseurs syntaxiques (`Yacc`/`Bison`) , avec quelques particularités
- Couplé à `ocamllex` pour réaliser la partie « front-end » d'un compilateur. Menhir liste les tokens à reconnaître par l'analyseur lexical
- Générateur d'analyse syntaxique LR(1) + actions associées aux productions. Les actions sont exécutées au moment des **réductions**.
- Les « actions » sont écrites en Ocaml.
- Gère une pile de « valeurs » en parallèle de la pile d'analyse syntaxique. Ces valeurs correspondent à un (unique) **attribut synthétisé**.

  Exemple : construction d'un AST représentant le programme à compiler

Schéma général



Première étape :
définir les nœuds du futur AST, *ie* des types `ocaml` (fichier **`ast.ml`**)

```
type opComp = Eq | Neq | Lt | Le | Gt | Ge

type expType =
  Id    of string
| Cste  of int
| Plus  of expType*expType
| Minus of expType*expType
| Times of expType*expType
| Div   of expType*expType
| Comp  of opComp*expType*expType
…
and instr =
  Ite of expType*instr*instr                    if_then_else
| Assign of string*expType                      affectation
type declarations = …
```

Structure d'un fichier .mly (1 / 4)

```
{
  (* code CAML copié en tête du fichier produit *)
open Ast
 …
}
%token <string> ID STRING
%token <int> CSTE
%token <Ast.opComp> RELOP
%token PLUS MINUS TIMES DIV
%token IF THEN ELSE
%token EOF

 %nonassoc RELOP  /* lowest precedence */
 %left PLUS MINUS
 %left TIMES DIV  /* highest precedence */
```

Type ocaml de la valeur du token

Indications de précédence et associativité

Structure d'un fichier .mly (2 / 4)

```
%type <expType> expr
%type <instr> instr
%type <decl> declaration
…
```

Types ocaml de la « valeur » associé à chaque non-terminal

Type de la valeur renvoyée par l'analyseur syntaxique

list: opérateur menhir. simplifie l'écriture des productions.

```
%start<Ast.progType> axiome
%%
axiome: ld = list(declaration) li = list(instr)  EOF
{
(* code ocaml pour cette production. Ici, on renvoie  l'AST du programme.
    ld:  l'AST de la partie déclarations, li: l'AST du corps du programme
*)
  (ld, li)
}
```

Structure d'un fichier .mly (3 / 4)

Le rôle principal des actions est de construire l'AST du programme

```
declaration:  …
{
  (* …  *)
}
```

syntaxe d'une déclaration et construction du fragment d'AST correspondant

La « valeur » d'un token sera fournie par l'analyse lexicale.

```
expr:
    x = ID                          { Id x }
  | v = CSTE                        { Cste v }
  | g = expr PLUS  d = expr         { Plus(g, d) }
  | g = expr TIMES d = expr         { Times(g, d) }
  …
  |  e = delimited(LPAREN, expr, RPAREN) { e }
```

Structure d’un fichier .mly (4 / 4)

```
instr: x = ID ASSIGN e = expr
      { Assign(x, e) }
|      IF si = expr THEN alors = instr ELSE sinon = instr
      { Ite(si, alors, sinon) }
```

Opérateurs de `menhir` et type de la valeur renvoyée

`list(`*X*`)`
`nonempty_list(`*X*`)`
`separated_list(`*séparateur*`, `*X*`)`
`separated_nonempty_list(`*séparateur*`, `*X*`)`
`delimited(`*opening*`, `*X*`, `*closing*`)`

} *type_de_X* `list`

`option(`*X*`)` — `Some` *type_de_X* `| None`

`boption(`*X*`)` — renvoie un booléen

`loption(`*X*`)` — renvoie une liste du type de *X*

Ces opérateurs se composent :

```
A : …   { renvoie une valeur de type monType }
LA : l = delimited(LPAREN,
                   separated_nonempty_list(COMMA, A)
                   RPAREN)
```

`l` sera une liste non vide de valeurs de type `monType`, donc `monType list`

Exercice: écrire les productions classiques pour `LA`

Usage « non standard » de Menhir :

- Menhir peut être utilisé en mode « interprète », ce qui peut aider à mettre au point une (sous-partie de) grammaire
- On peut alors appeler l'analyseur syntaxique interactivement sur un « mot » du langage, **menhir** imprimant une version indentée de son « arbre syntaxique »
- Le mot entré doit tenir sur une ligne et ne contient que des **tokens** (*ie on doit simuler l'analyseur lexical)*
- *Syntaxe d'appel :*

  menhir –interpret –interpret-show-cst *fichier.mly*

Usage « non standard » de Menhir (exemple)

```
%token PLUS TIMES LPAR RPAR EOF
%token <int> CONST
%token <string> ID

%left PLUS
%left TIMES

%start <unit> main
%%
main:
| expression EOF { }

expression:
| expression PLUS expression  { }
| expression TIMES expression { }
| LPAR expression RPAR      { }
| CONST                 { }
| ID                  { }
```

Usage « non standard » de Menhir (suite)

```
menhir –interpret –interpret-show-cst expr.mlyD
Ready !
ID PLUS CONST TIMES ID PLUS
ACCEPT
[main:
  [expression:
    [expression:
      [expression: ID]
      PLUS
      [expression: [expression: CONST] TIMES [expression: ID]]
    ]
    PLUS
    [expression: ID]
  ]
  EOF
]
```

OCAMLLEX

- Principe similaire à d'autres générateurs d'analyseurs lexicaux (`Flex`) :

  On fournit des expressions régulières ; pour chacune, on construit un AFD qui reconnaît le langage défini par l'expression

  On construit l'AFD minimal pour l'union de ces AFD

  Le mot en entrée est découpé en une séquence de sous-mots dont chacun est reconnu par un des AFD

- `ocamllex` fournit :

  Un mécanisme pour définir des « abréviations »

  Des opérateurs pour les expressions régulières

  Des stratégies pour les ambiguïtés dans la reconnaissance d'un token

  Un mécanisme pour gérer les positions dans le texte

- Couplé à `menhir` pour réaliser la partie « front-end » d'un compilateur :

  menhir appelle l'analyseur lexical quand il a besoin du prochain token

Particularités de OCAMLLEX

- On regroupe un ensemble d'expressions régulières dans un sous-analyseur
- À chaque sous-analyseur correspond une fonction Ocaml
- Ces fonctions peuvent être (mutuellement) récursives

  On a donc un mécanisme qui dépasse ce qui est reconnaissable par AFD (*par exemple les commentaires emboîtés*)

  Il peut être plus simple d'écrire séparément des sous-analyseurs spécialisés : pour les commentaires, pour les string, pour les cas standard

- Une de ces fonctions produites par `ocamllex` sert de point d'appel pour Menhir ; chacun de ces appels doit renvoyer le prochain token

Extrait du fichier **`tpParse.mly`** (rappel !)

```
%token <string> ID STRING
%token <int> CSTE
%token <Ast.opComp> RELOP
%token PLUS MINUS TIMES DIV
%token LPAREN RPAREN
%token SEMICOLON COMMA
%token IF THEN ELSE
%token UMINUS
%token EOF
```

Type ocaml de la valeur lexicale du token

« pseudo » token

« End of file »

Extrait du fichier **`tpParse.mli`** produit automatiquement par `menhir`.
Ce sont les tokens que votre analyseur doit reconnaître !

```
type token =
| UMINUS
| TIMES
| THEN
| STRING of (string)
| SEMICOLON
| RPAREN
| RELOP of (Ast.opComp)
| PLUS
| MINUS
| LPAREN
| IF
| ID of (string)
| EOF
| ELSE
| DIV
| CSTE of (int)
| COMMA


exception Error
```

Type « construit » ocaml :
chaque constructeur correspond à un token
Pour les tokens « avec valeur » on spécifie le type de cette dernière

Structure d'un fichier .mll (début)

```
{ (* code CAML qui sera copié en tête du fichier produit *)
open Ast        (* module associé au fichier ast.ml *)
open TpParse    (* produit par menhir *)
open Lexing     (* dans la bibliothèque standard *)

(* gerer les numeros de ligne + position dans la ligne *)
let next_line lexbuf = Lexing.new_line lexbuf

(* ici votre code pour différencier mots-clefs et idents *)
 …
}

let Lettre = ['a'-'z' 'A'-'Z']          (* abréviations *)
let Chiffre = ['0'-'9']
let LC = (Lettre | Chiffre)
```

Structure d'un fichier .mll (suite)

Nom de la fonction ocaml de cet analyseur

ici, il faudrait distinguer mots-clef et idents

```
rule
token = parse
  Lettre LC* as id        { ID id }
| [' ''\t''\r']           { token lexbuf }
| '\n'                    { next_line lexbuf; token lexbuf}
| Chiffre+ as lxm         { CSTE(int_of_string lxm) }
| "/*"                    { comment lexbuf }
| '+'                     { PLUS }
  ...
| "<"                     { RELOP (Ast.Lt) }
| "<="                    { RELOP (Ast.Le) }
| eof                     { EOF }
and
comment = parse
  "*/"                    { token lexbuf }
| '\n'                    { new_line lexbuf; comment lexbuf }
  ...
|  _                      { comment lexbuf }
```

Autre fonction ocaml

Appel récursif pour gérer la suite du commentaire

Opérateurs `ocamllex` pour les expressions régulières :

voir la doc en ligne: *union, concaténation, étoile, optionnel, …*

Voir aussi : https://dev.realworldocaml.org/parsing-with-ocamllex-and-menhir.html

Stratégies pour résoudre les conflits de reconnaissance

- On privilégie l'expression qui reconnaît la portion de texte **la plus longue** possible
- **À longueurs égales**, on privilégie l'expression qui apparaît **en premier** dans votre fichier `.mll`
- Le « joker » `_` matche n'importe quel caractère. Attention à son usage : doit servir à reconnaître ce qui n'est pas reconnu par ailleurs !

`ocamllex` tpLex.mll va ici produire le fichier `tpLex.ml` qui contient deux fonctions `TpLex.comment` et **`TpLex.token`**.

La fonction `TpLex.token est` à fournir en argument à votre fonction d'analyse syntaxique (voir le code dans le fichier `main.ml`)qui sera fourni pour le TP/projet)