

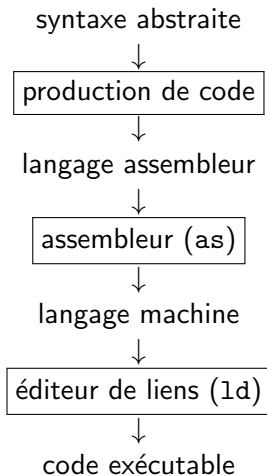
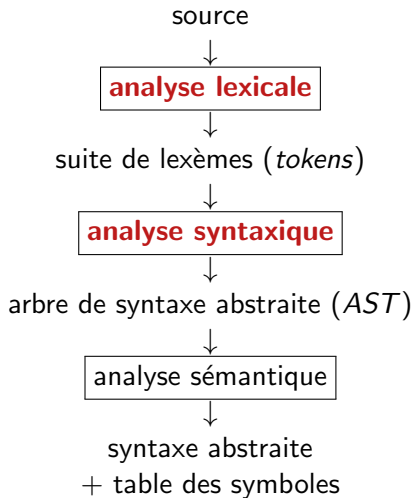
Université Paris-Sud

# Compilation et langages

Thibaut Balabonski  
d'après Jean-Christophe Filliâtre @ ENS

Cours 2 / 23 septembre 2016

Analyse **lexicale** avec ocamllex  
Analyse **syntaxique** avec menhir



---

# Analyse lexicale

Quand j'étais enfant, on m'avait dit que le Père Noël descendait par la cheminée, et que les ordinateurs se programmaient en binaire. J'ai appris depuis que la programmation se faisait de préférence dans des langages de haut niveau, plus abstraits et plus expressifs.

Quand j'étais enfant, on m'avait dit que le Père Noël descendait par la cheminée, et que les ordinateurs se programmaient en binaire. J'ai appris depuis que la programmation se faisait de préférence dans des langages de haut niveau, plus abstraits et plus expressifs.

*introduction de la thèse de X. Leroy*

L'analyse lexicale est le découpage du texte source en « mots »

De même que dans les langues naturelles, ce découpage en mots facilite le travail de la phase suivante, l'analyse syntaxique

Ces mots sont appelés des **lexèmes** (*tokens*)

source = suite de caractères

```
fun x -> (* ma fonction *)  
  x+1
```

↓  
**analyse lexicale**

↓  
suite de lexèmes

fun	x	->	x	+	1
-----	---	----	---	---	---

↓

↓  
**analyse syntaxique**  
↓

syntaxe abstraite

```
graph TD
    Fun[Fun] --- X["x"]
    Fun --- App1[App]
    App1 --- App2[App]
    App1 --- Const1[Const]
    App2 --- Op[Op]
    App2 --- Var[Var]
    Const1 --- 1[1]
    Op --- Plus["+"]
    Var --- X2["x"]
```



Les blancs (espace, retour chariot, tabulation, etc.) jouent un rôle dans l'analyse lexicale ; ils permettent notamment de séparer deux lexèmes

Ainsi `funx` est compris comme un seul lexème (l'identificateur `funx`) et `fun x` est compris comme deux lexèmes (le mot clé `fun` et l'identificateur `x`)

De nombreux blancs sont néanmoins inutiles (comme dans `x + 1`) et simplement ignorés

Les blancs n'apparaissent pas dans le flot de lexèmes renvoyé

Les conventions diffèrent selon les langages,  
et certains des caractères « blancs » peuvent être significatifs

Exemples :

- les tabulations pour `make`
- retours chariot et espaces de début de ligne en Python ou en Haskell (l'indentation détermine la structure des blocs)

Les commentaires jouent le rôle de blancs

```
fun(* et hop *)x -> x + (* j'ajoute un *) 1
```

Ici le commentaire `(* et hop *)` joue le rôle d'un blanc significatif (sépare deux lexèmes) et le commentaire `(* j'ajoute un *)` celui d'un blanc inutile

Note : les commentaires sont parfois exploités par certains outils (ocamldoc, javadoc, etc.), qui les traitent alors différemment dans leur propre analyse lexicale

```
val length : 'a list -> int  
(** Return the length (number of elements) of ...
```

Pour réaliser l'analyse lexicale, on va utiliser

- des **expressions régulières** pour décrire les lexèmes
- des **automates finis** pour les reconnaître

On exploite notamment la capacité à construire automatiquement un automate fini déterministe reconnaissant le langage décrit par une expression régulière (cf. cours langages formels)

---

# Expressions régulières

$r ::=$	$\emptyset$	langage vide
	$  \epsilon$	mot vide
	$  a$	caractère $a \in A$
	$  r r$	concaténation
	$  r   r$	alternative
	$  r \star$	étoile

Conventions : l'étoile a la priorité la plus forte, puis la concaténation, puis enfin l'alternative

Le **langage** défini par l'expression régulière  $r$  est l'ensemble de mots  $L(r)$  défini par

$$L(\emptyset) = \emptyset$$

$$L(\epsilon) = \{\epsilon\}$$

$$L(a) = \{a\}$$

$$L(r_1 r_2) = \{w_1 w_2 \mid w_1 \in L(r_1) \wedge w_2 \in L(r_2)\}$$

$$L(r_1 \mid r_2) = L(r_1) \cup L(r_2)$$

$$L(r^*) = \bigcup_{n \geq 0} L(r^n) \quad \text{où } r^0 = \epsilon, \quad r^{n+1} = r r^n$$

Ces langages sont aussi reconnus par des **automates finis**.

Sur l'alphabet  $\{a, b\}$

- mots de trois lettres

$$(a|b)(a|b)(a|b)$$

- mots se terminant par un  $a$

$$(a|b) \star a$$

- mots alternant  $a$  et  $b$

$$(b|\epsilon)(ab) \star (a|\epsilon)$$

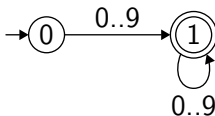


lexème	expression régulière	automate
mot clé fun	<i>f u n</i>	<pre> graph LR     0((0)) -- f --&gt; 1((1))     1 -- u --&gt; 2((2))     2 -- n --&gt; 3(((3)))             </pre>
symbole +	+	<pre> graph LR     0((0)) -- + --&gt; 1(((1)))             </pre>
symbole ->	->	<pre> graph LR     0((0)) -- - --&gt; 1((1))     1 -- &gt; --&gt; 2(((2)))             </pre>

Expression régulière

$(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$

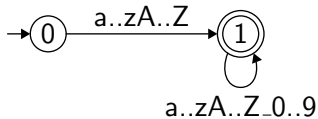
Automate



Expression régulière

$$(a|b|\dots|z|A|B|\dots|Z)(a|b|\dots|z|A|B|\dots|Z|_0|1|\dots|9)^{\star}$$

Automate

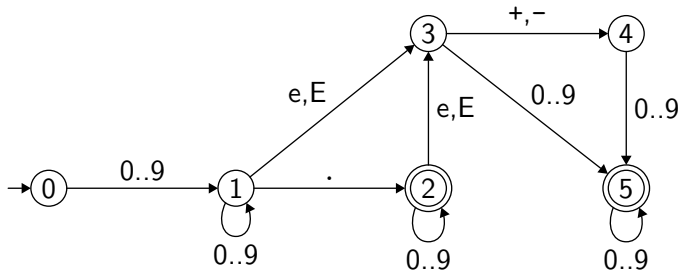


Expression régulière

$$d d^* (. d^* | (\epsilon | . d^*) (e | E) (\epsilon | + | -) d d^*)$$

où  $d = 0|1|\dots|9$

Automate



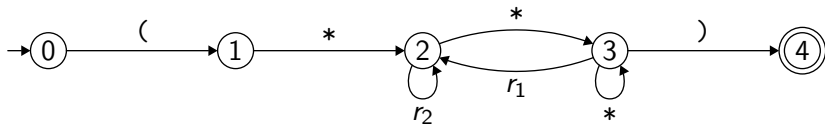
## Expression régulière

$$\boxed{(} \boxed{*} \left( \boxed{*} * r_1 \mid r_2 \right) * \boxed{*} \boxed{*} * \boxed{)}$$

où  $r_1$  = tous les caractères sauf \* et )

et  $r_2$  = tous les caractères sauf \*

## Automate fini



---

# Analyseur lexical

Un **analyseur lexical** est un automate fini pour la « réunion » de toutes les expressions régulières définissant les lexèmes

Le fonctionnement de l'analyseur lexical, cependant, est différent de la simple reconnaissance d'un mot par un automate, car

- il faut décomposer un mot (le source) en une **suite** de mots reconnus
- il peut y avoir des **ambiguïtés**
- il faut construire les lexèmes (les états finaux contiennent des **actions**)

Le mot `funx` est reconnu par l'expression régulière des identificateurs, mais contient un préfixe reconnu par une autre expression régulière (`fun`)

⇒ On fait le choix de reconnaître le lexème le plus long possible

Le mot `fun` est reconnu par l'expression régulière du mot clé `fun` mais aussi par celle des identificateurs

⇒ On classe les lexèmes par ordre de priorité



Avec les trois expressions régulières

$a, \quad ab, \quad bc$

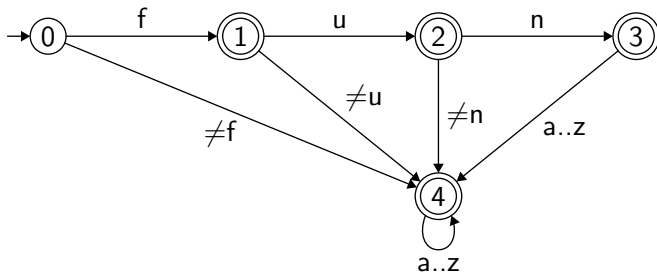
un analyseur lexical va **échouer** sur l'entrée

$abc$

( $ab$  est reconnu, comme plus long, puis échec sur  $c$ )

pourtant le mot  $abc$  appartient au langage  $a|ab|bc$

Exemple : le mot clé fun et les identificateurs



L'analyseur lexical doit donc mémoriser le dernier état final rencontré, le cas échéant

Lorsqu'il n'y a plus de transition possible, deux possibilités :

- aucune position finale n'a été mémorisée  
⇒ échec de l'analyse lexicale
- on a lu le préfixe  $wv$  de l'entrée, avec  $w$  le lexème reconnu par le dernier état final rencontré  
⇒ on renvoie le lexème  $w$ , et l'analyse redémarre avec  $v$  préfixé au reste de l'entrée

En pratique, on dispose d'outils qui construisent les analyseurs lexicaux à partir de leur description par des expressions régulières et des actions

C'est la grande famille de **lex** : `lex`, `flex`, `jflex`, `ocamllex`, etc.

---

# L'outil `ocamllex`

Un fichier `ocamllex` porte le suffixe `.mll` et a la forme suivante

```
{  
  ... code OCaml arbitraire ...  
}  
rule f1 = parse  
| regexp1 { action1 }  
| regexp2 { action2 }  
| ...  
and f2 = parse  
  ...  
and fn = parse  
  ...  
{  
  ... code OCaml arbitraire ...  
}
```

On compile le fichier `lexer.mll` avec `ocamllex`

```
% ocamllex lexer.mll
```

Ce qui produit un fichier OCaml `lexer.ml` qui définit une fonction pour chaque analyseur `f1`, ..., `fn` :

```
val f1 : Lexing.lexbuf -> type1  
val f2 : Lexing.lexbuf -> type2  
...  
val fn : Lexing.lexbuf -> typen
```

Le type Lexing.lexbuf est celui de la structure de données qui contient l'état d'un analyseur lexical

Le module Lexing de la bibliothèque standard fournit plusieurs moyens de construire une valeur de ce type

```
val from_channel : Pervasives.in_channel -> lexbuf
```

```
val from_string : string -> lexbuf
```

```
val from_function : (string -> int -> int) -> lexbuf
```



# Les expressions régulières d'ocamllex

<code>_</code>	n'importe quel caractère
<code>'a'</code>	le caractère <code>'a'</code>
<code>"foobar"</code>	la chaîne <code>"foobar"</code> (en particulier $\epsilon = ""$ )
<code>[caractères]</code>	ensemble de caractères (par ex. <code>['a'-'z' 'A'-'Z']</code> )
<code>[^caractères]</code>	complémentaire (par ex. <code>[^ '"']</code> )
<code><math>r_1 \mid r_2</math></code>	l'alternative
<code><math>r_1 r_2</math></code>	la concaténation
<code><math>r^*</math></code>	l'étoile
<code><math>r^+</math></code>	une ou plusieurs répétitions de $r$ ( $\stackrel{\text{def}}{=} r r^*$ )
<code><math>r^?</math></code>	une ou zéro occurrence de $r$ ( $\stackrel{\text{def}}{=} \epsilon \mid r$ )
<code>eof</code>	la fin de l'entrée

## Identificateurs

```
| ['a'-'z' 'A'-'Z'] ['a'-'z' 'A'-'Z' '_' '0'-'9']* { ... }
```

## Constantes entières

```
| ['0'-'9']+ { ... }
```

## Constantes flottantes

```
| ['0'-'9']+  
  ( '.' ['0'-'9']*  
    | ('.' ['0'-'9']*)? ['e' 'E'] ['+' '-']? ['0'-'9']+ )  
{ ... }
```

On peut définir des raccourcis pour des expressions régulières

```
let letter = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']
let decimals = '.' digit*
let exponent = ['e' 'E'] ['+' '-']? digit+

rule token = parse
  | letter (letter | digit | '_')*           { ... }
  | digit+                                   { ... }
  | digit+ (decimals | decimals? exponent) { ... }
```

Pour les analyseurs définis avec le mot clé `parse`, la règle du plus long lexème reconnu s'applique

À longueur égale, c'est la règle qui apparaît en premier qui l'emporte

```
| "fun"           { Tfun }  
| ['a'-'z']+ as s { Tident s }
```

Pour le plus court, il suffit d'utiliser `shortest` à la place de `parse`

```
rule scan = shortest  
| regexp1 { action1 }  
| regexp2 { action2 }  
...
```

On peut nommer la chaîne reconnue, ou des sous-chaînes reconnues par des sous-expressions régulières, à l'aide de la construction **as**

```
| ['a'-'z']+ as s { ... }
```

```
| (['+' '-' ]? as sign) (['0'-'9']+ as num) { ... }
```

Dans une action, il est possible de rappeler récursivement l'analyseur lexical, ou l'un des autres analyseurs simultanément définis

Le tampon d'analyse lexical doit être passé en argument ;  
il est contenu dans la variable `lexbuf`

Il est ainsi très facile de traiter les blancs :

```
rule token = parse
| [ ' ' '\t' '\n' ]+ { token lexbuf }
| ...
```

Pour traiter les commentaires, on peut utiliser une expression régulière

... ou un analyseur dédié :

```
rule token = parse
| "(" { comment lexbuf }
| ...

and comment = parse
| "*)" { token lexbuf }
| _ { comment lexbuf }
| eof { failwith "commentaire non terminé" }
```

Avantage : on traite correctement l'erreur liée à un commentaire non fermé

Autre intérêt : on traite facilement les **commentaires imbriqués**

Avec un compteur

```
rule token = parse
| "(" { level := 1; comment lexbuf; token lexbuf }
| ...

and comment = parse
| "*)" { decr level; if !level > 0 then comment lexbuf }
| "(" { incr level; comment lexbuf }
| _ { comment lexbuf }
| eof { failwith "commentaire non terminé" }
```



...ou sans compteur !

```
rule token = parse
| "(" { comment lexbuf; token lexbuf }
| ...

and comment = parse
| "*)" { () }
| "(" { comment lexbuf; comment lexbuf }
| _ { comment lexbuf }
| eof { failwith "commentaire non terminé" }
```

note : on a donc dépassé la puissance des expressions régulières

Quatre « règles » à ne pas oublier quand on écrit un analyseur lexical

1. traiter les **blancs**
2. les règles **les plus prioritaires en premier** (par ex. mots clés avant identificateurs)
3. signaler les **erreurs lexicales** (caractères illégaux, mais aussi commentaires ou chaînes non fermés, séquences d'échappement illégales, etc.)
4. traiter la **fin de l'entrée** (eof)

Par défaut, `ocamllex` encode l'automate dans une **table**, qui est interprétée à l'exécution

L'option `-ml` permet de produire du code OCaml pur, où l'automate est encodé par des fonctions ; ce n'est pas recommandé en pratique cependant

Même en utilisant une table, l'automate peut prendre beaucoup de place, en particulier s'il y a de nombreux mots clés dans le langage

Il est préférable d'utiliser une seule expression régulière pour les identificateurs et les mots clés, et de les séparer ensuite grâce à une table des mots clés

```
{  
  let keywords = Hashtbl.create 97  
  let () = List.iter (fun s -> Hashtbl.add keywords s ())  
    ["and", AND; "as", AS; "assert", ASSERT;  
     "begin", BEGIN; ...  
}  
rule token = parse  
  | ident as s  
  { try Hashtbl.find keywords s with Not_found -> IDENT s }
```

Si on souhaite un analyseur lexical qui ne soit pas sensible à la casse, surtout ne pas écrire

```
| ("a"|"A") ("n"|"N") ("d"|"D")  
  { AND }  
| ("a"|"A") ("s"|"S")  
  { AS }  
| ("a"|"A") ("s"|"S") ("s"|"S") ("e"|"E") ("r"|"R") ("t"|"T")  
  { ASSERT }  
| ...
```

mais plutôt

```
rule token = parse  
  | ident as s  
    { let s = String.lowercase s in  
      try Hashtbl.find keywords s with Not_found -> IDENT s }
```

Pour compiler (ou recompiler) les modules OCaml, il faut déterminer les **dépendances** entre ces modules, grâce à `ocamldep`

Or `ocamldep` ne connaît pas la syntaxe `ocamllex`  $\Rightarrow$  il faut donc s'assurer de la fabrication préalable du code OCaml avec `ocamllex`

le Makefile ressemble donc à ceci (cf. TP) :

```
lexer.ml: lexer.mll
    ocamllex lexer.mll

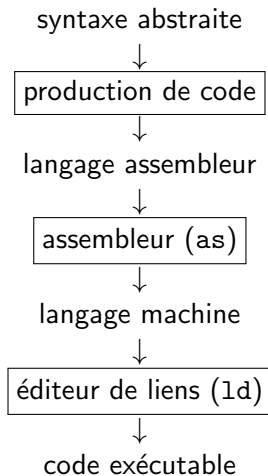
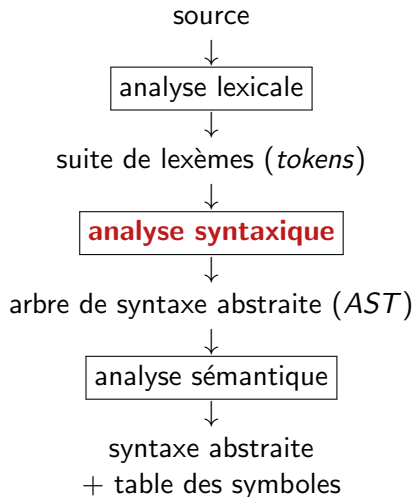
.depend: lexer.ml
    ocamldep *.ml *.mli > .depend

include .depend
```

**Alternative :** utiliser `ocamlbuild`

- les **expressions régulières** sont à la base de l'analyse lexicale
- le travail est grandement automatisé par des outils tels que **ocamllex**
- **ocamllex** est plus expressif que les expressions régulières, et peut être utilisé bien au delà de l'analyse lexicale

(note : ces propres transparents sont réalisés avec un préprocesseur  $\text{\LaTeX}$  écrit à l'aide d'**ocamllex**)





---

# Syntaxe abstraite

Un programme, en tant qu'objet syntaxique (suite de caractères), est trop difficile à manipuler. On préfère utiliser la **syntaxe abstraite**  
Ainsi, les expressions :

$2*(k+1)$

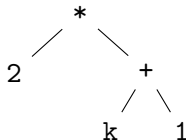
et

$(2 * ((k) + 1))$

et

$2 * (* \text{ je multiplie par deux } *) (k + 1)$

représentent toutes le même **arbre de syntaxe abstraite**



On définit une syntaxe abstraite par une grammaire, de la manière suivante

$e$	$::=$	$c$	constante
		$x$	variable
		$e + e$	addition
		$e \times e$	multiplication
		$\dots$	

Traduction : « une expression  $e$  est

- soit une constante,
- soit une variable,
- soit l'addition de deux expressions,
- soit la multiplication de deux expressions,
- etc. »

La notation  $e_1 + e_2$  de cette syntaxe abstraite emprunte le symbole de la syntaxe concrète

Mais on aurait pu tout aussi bien choisir  $Add(e_1, e_2)$  ou  $+(e_1, e_2)$ , etc.

En Caml, on réalise les arbres de syntaxe abstraite par des types récurifs

```
type binop = Plus | Mult | ...  
  
type expr =  
  | Eint    of int  
  | Evar    of string  
  | Ebinop  of binop * expr * expr
```

L'expression  $2 * (k + 1)$  est représentée par

```
Ebinop (Mult, Eint 2, Ebinop(Plus, Evar "k", Eint 1))
```

On appelle **sucre syntaxique** une construction de la syntaxe concrète qui n'existe pas dans la syntaxe abstraite

Elle est donc traduite à l'aide d'autres constructions de la syntaxe abstraite, généralement pendant l'analyse syntaxique

Exemple : en Caml l'expression  $[e_1; e_2; \dots; e_n]$  est du sucre pour

$$e_1 :: e_2 :: \dots :: e_n :: []$$

---

# Analyse syntaxique

L'objectif de l'analyse syntaxique est de reconnaître les phrases appartenant à la syntaxe du langage

Son entrée est le flot des lexèmes construits par l'analyse lexicale, sa sortie est un arbre de syntaxe abstraite



suite de lexèmes

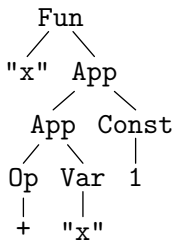
fun x -> ( x + 1 )



**analyse syntaxique**



syntaxe abstraite



En particulier, l'analyse syntaxique doit détecter les erreurs de syntaxe et

- les localiser précisément
- les identifier (le plus souvent seulement « erreur de syntaxe » mais aussi « parenthèse non fermée », etc.)
- voire, reprendre l'analyse pour découvrir de nouvelles erreurs

Pour l'analyse syntaxique, on va utiliser

- une **grammaire non contextuelle** pour décrire la syntaxe
- un **automate à pile** pour la reconnaître

C'est l'analogue des expressions régulières / automates finis utilisés dans l'analyse lexicale

## Définition

*Une grammaire non contextuelle (ou hors contexte) est un quadruplet  $(N, T, S, R)$  où*

- *$N$  est un ensemble fini de **symboles non terminaux***
- *$T$  est un ensemble fini de **symboles terminaux***
- *$S \in N$  est le symbole de départ (dit **axiome**)*
- *$R \subseteq N \times (N \cup T)^*$  est un ensemble fini de **règles de production***

## Exemple : expressions arithmétiques

- Symboles terminaux : +, \*, (, ), int
- Symboles non terminaux :  $E$
- Symbole de départ :  $E$
- Règles :
$$\begin{array}{lcl} E & \rightarrow & E + E \\ & & | \quad E * E \\ & & | \quad ( E ) \\ & & | \quad \text{int} \end{array}$$

int désigne ici le lexème correspondant à une constante entière (*i.e.* sa nature, pas sa valeur)

## Définition

Un mot  $u \in (N \cup T)^*$  se **dérive** en un mot  $v \in (N \cup T)^*$ , et on note  $u \rightarrow v$ , s'il existe une décomposition

$$u = u_1 X u_2$$

avec  $X \in N$ ,  $X \rightarrow \beta \in R$  et

$$v = u_1 \beta u_2$$

Exemple :

$$\underbrace{E *}_{u_1} \left( \underbrace{E}_X \right) \underbrace{ )}_{u_2} \rightarrow E * \left( \underbrace{E + E}_{\beta} \right)$$

Une suite  $w_1 \rightarrow w_2 \rightarrow \dots \rightarrow w_n$  est appelée une dérivation

On parle de **dérivation gauche** (resp. **droite**) si le non terminal réduit est systématiquement le plus à gauche *i.e.*  $u_1 \in T^*$  (resp. le plus à droite *i.e.*  $u_2 \in T^*$ )

On note  $\rightarrow^*$  la clôture réflexive transitive de  $\rightarrow$

$$\begin{aligned} E &\rightarrow E * E \\ &\rightarrow \text{int} * E \\ &\rightarrow \text{int} * ( E ) \\ &\rightarrow \text{int} * ( E + E ) \\ &\rightarrow \text{int} * ( \text{int} + E ) \\ &\rightarrow \text{int} * ( \text{int} + \text{int} ) \end{aligned}$$

En particulier

$$E \rightarrow^* \text{int} * ( \text{int} + \text{int} )$$



## Définition

À toute dérivation  $S \rightarrow^* w$ , on peut associer un **arbre de dérivation**, dont les nœuds sont étiquetés ainsi

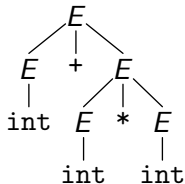
- la racine est  $S$
- les feuilles forment le mot  $w$  dans l'ordre infixe
- tout nœud interne  $X$  est un non terminal dont les fils sont étiquetés par  $\beta \in (N \cup T)^*$  avec  $X \rightarrow \beta$  une règle de la dérivation

Attention : ce n'est pas la même chose que l'arbre de syntaxe abstraite

La dérivation gauche

$$E \rightarrow E + E \rightarrow \text{int} + E \rightarrow \text{int} + E * E \rightarrow \text{int} + \text{int} * E \rightarrow \text{int} + \text{int} * \text{int}$$

donne l'arbre de dérivation



mais la dérivation droite

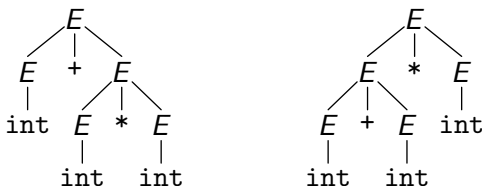
$$E \rightarrow E + E \rightarrow E + E * E \rightarrow E + E * \text{int} \rightarrow E + \text{int} * \text{int} \rightarrow \text{int} + \text{int} * \text{int}$$

également

## Définition

Une grammaire est dite **ambiguë** si un mot au moins admet plusieurs arbres de dérivation

exemple : le mot `int + int * int` admet les deux arbres de dérivations



Désambiguïsation : modification de la grammaire ou ajout de priorités  
 $\Rightarrow$  on en parle la semaine prochaine

Nous verrons la semaine prochaine comment cette analyse est faite

En pratique, le travail est automatisé par de nombreux outils qui construisent des analyseurs syntaxiques à partir d'une grammaire et d'actions

C'est la grande famille de **yacc**, bison, ocaml yacc, cups, menhir, ...  
(YACC signifie *Yet Another Compiler Compiler*)

---

# L'outil Menhir

Menhir est un outil qui transforme une grammaire en un analyseur OCaml

Chaque production de la grammaire est accompagnée d'une **action sémantique** *i.e.* du code OCaml construisant une valeur sémantique (typiquement un arbre de syntaxe abstraite)

Menhir s'utilise conjointement avec un analyseur lexical (tel `ocamllex`)

Un fichier Menhir porte le suffixe `.mly` et a la structure suivante

```
%{  
    ... code OCaml arbitraire ...  
%}  
...déclaration des tokens...  
...déclaration des précédences et associativités...  
...déclaration des points d'entrée...  
%%  
non-terminal-1:  
| production { action }  
| production { action }  
;  
  
non-terminal-2:  
| production { action }  
...  
%%  
    ... code OCaml arbitraire ...
```

```
%token PLUS LPAR RPAR EOF
```

```
%token <int> INT
```

```
%start <int> phrase
```

```
%%
```

```
phrase:
```

```
    e = expression; EOF  { e }
```

```
;
```

```
expression:
```

```
    | e1 = expression; PLUS; e2 = expression { e1 + e2 }
```

```
    | LPAR; e = expression; RPAR              { e }
```

```
    | i = INT                                  { i }
```

```
;
```



On compile le fichier `arith.mly` de la manière suivante

```
% menhir -v arith.mly
```

On obtient du code OCaml pur dans `arith.ml(i)`, qui contient notamment

- La déclaration d'un type `token`

```
type token = RPAR | PLUS | LPAR | INT of int | EOF
```

- et, pour le non terminal déclaré avec `%start`, une fonction du type

```
val phrase: (Lexing.lexbuf -> token) -> Lexing.lexbuf -> int
```

Comme on le voit, cette fonction prend en argument un analyseur lexical, du type de celui produit par `ocamllex` (cf. début du cours)

Lorsque la grammaire est ambiguë, Menhir présente les **conflits** à l'utilisateur

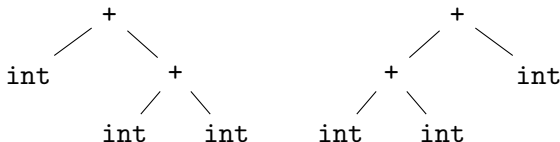
- les fichiers `.automaton` et `.conflicts` contiennent, le cas échéant, des informations sur les conflits détectés
- nous aborderons ce point en détail la semaine prochaine

Sur la grammaire ci-dessus, Menhir signale un conflit

```
% menhir -v arith.mly  
Warning: one state has shift/reduce conflicts.  
Warning: one shift/reduce conflict was arbitrarily resolved.
```

qui concerne l'ambiguïté entre deux lectures possibles de l'expression

1 + 2 + 3



On peut résoudre ce conflit en indiquant que PLUS est associatif à gauche

```
%token PLUS LPAR RPAR EOF
%token <int> INT
%left PLUS
%start <int> phrase
%%
phrase:
    e = expression; EOF { e }
;
expression:
| e1 = expression; PLUS; e2 = expression { e1 + e2 }
| LPAR; e = expression; RPAR { e }
| i = INT { i }
;
```

Pour les conflits entre différents opérateurs, on peut aussi associer des priorités aux lexèmes, avec la convention suivante :

- l'ordre de déclaration des associativités fixe les priorités (les premiers lexèmes ont les priorités les plus faibles)
- plusieurs lexèmes peuvent apparaître sur la même ligne, ayant ainsi la même priorité

Exemple :

```
%left PLUS MINUS  
%left TIMES DIV
```

Pour que les phases suivantes de l'analyse (typiquement le typage) puissent **localiser** les messages d'erreur, il convient de conserver une information de localisation dans l'arbre de syntaxe abstraite

Menhir fournit cette information dans `$startpos` et `$endpos`, deux valeurs du type `Lexing.position`; cette information lui a été transmise par l'analyseur lexical

Attention : `ocamllex` ne maintient automatiquement que la position absolue dans le fichier; pour avoir les numéros de ligne et de colonnes à jour, il faut un traitement spécial du retour chariot dans l'analyseur lexical (voir par exemple `lexer.mll` fourni dans le TP)

Une façon de conserver l'information de localisation dans l'arbre de syntaxe abstraite est la suivante

```
type expression =  
  { desc: desc;  
    loc : Lexing.position * Lexing.position }  
  
and desc =  
  | Econst of int  
  | Eplus  of expression * expression  
  | Eneg   of expression  
  | ...
```

La grammaire peut donc ressembler à ceci

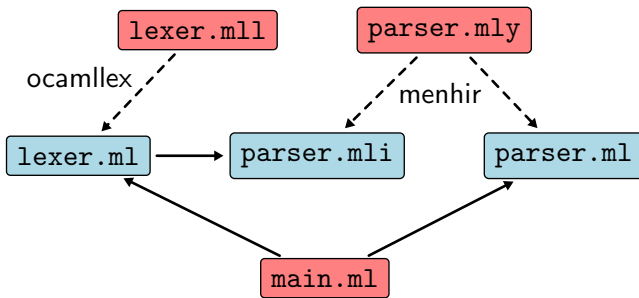
```
expression:
```

```
| d = desc { { desc = d; loc = $startpos, $endpos } }  
;
```

```
desc:
```

```
| i = INT { Econst i }  
| e1 = expression; PLUS; e2 = expression { Eplus (e1, e2) }  
| ...
```





= source utilisateur

= construit automatiquement

→  = dépendance

Comme dans le cas d'ocamllex, il faut s'assurer de l'application de menhir avant le calcul des dépendances

Le Makefile ressemble donc à ceci :

```
lexer.ml: lexer.mll
    ocamllex lexer.mll

parser.mli parser.ml: parser.mly
    menhir -v parser.mly

.depend: lexer.ml parser.mli parser.ml
    ocamldep *.ml *.mli > .depend

include .depend
```

(cf. Makefile fourni avec le TP)

- les **grammaires non contextuelles** sont à la base de l'analyse syntaxique
- le travail est grandement automatisé par des outils tels que **menhir**
- la détection et la résolution des ambiguïtés est un point délicat  
⇒ patientez une semaine