Rappels et compléments de compilation
Analyses lexicale et syntaxique
Cours de Compilation Avancée (4I504)

Benjamin Canou
Université Pierre et Marie Curie

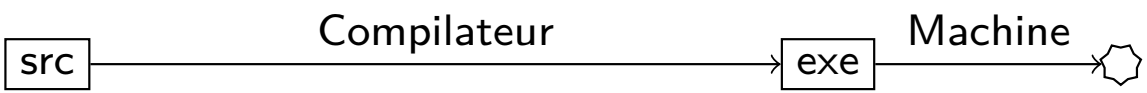
Année 2016/2017 – Semaine 1

Rappels

Qu'est-ce que la compilation ?

Principe de base de la compilation :

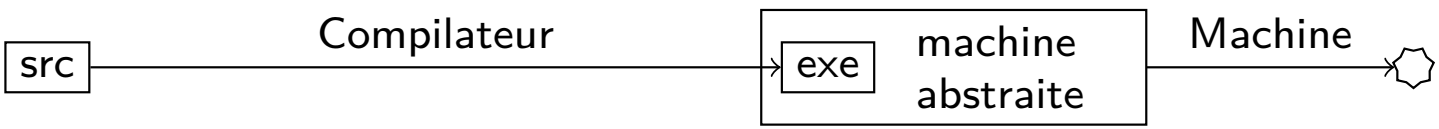
Traduction du **code source** vers du **code machine** (*code natif*).



Qu'est-ce que la compilation ?

Compilation pour une machine virtuelle (*VM*):

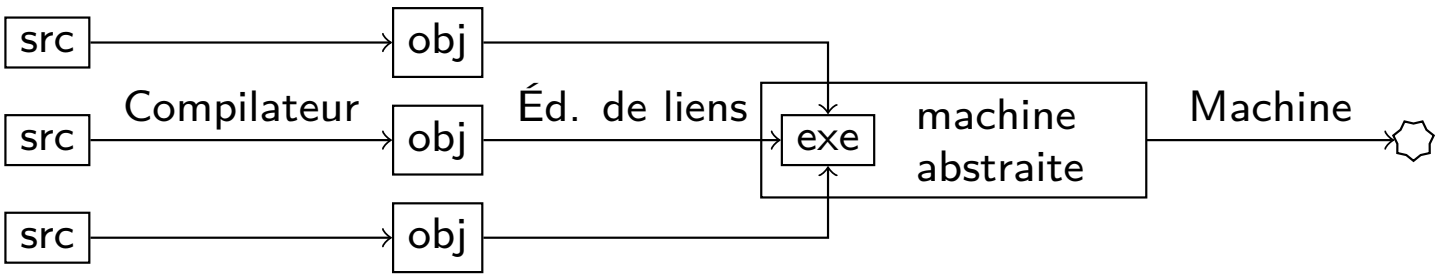
Production de **code-octet *(bytecode)*** interprété ou compilé à la volée vers du code machine.



Qu'est-ce que la compilation ?

Compilation séparée :

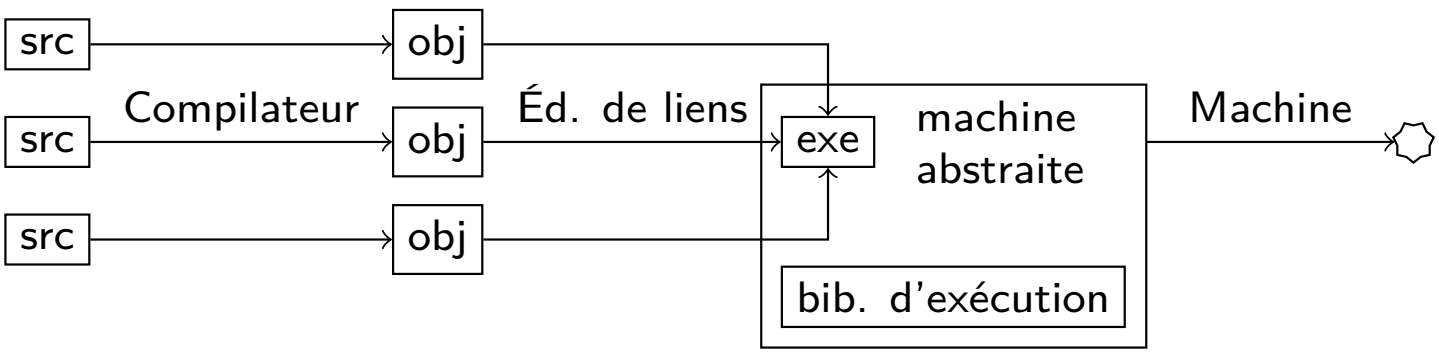
Liaison de **fichiers objet**, un fichier objet par **unité de compilation** du langage (*classe*, *module*, *package*, *etc.*).



Qu'est-ce que la compilation ?

Utilisation d'une **bibliothèque d'exécution** (*runtime*) :
pour le support des langages de haut niveau (*gestion mémoire, entrées/sorties, chargement dynamique, appels de méthods, continuations, etc.*).

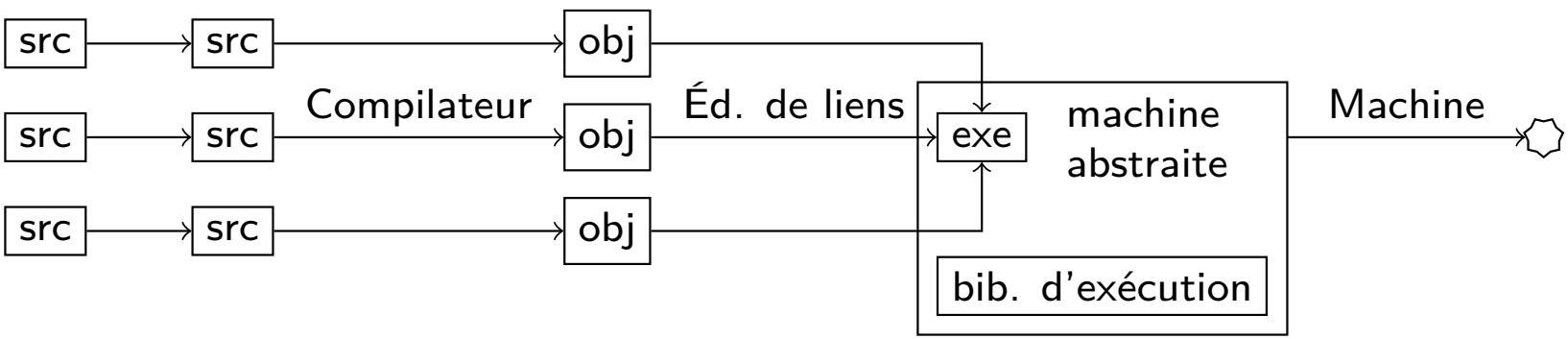
- VM : intégré dans la machine virtuelle
- Code natif : lié dans l'exécutable



Qu'est-ce que la compilation ?

Transformations source-à-source :

- Préprocesseur : même langage de sortie, pour nettoyer, appliquer des macros, etc.
- Traduction (compilation) : utilisation d'un langage existant comme cible. Comme pour une VM, il faut éventuellement une bibliothèque d'éxécution.



Chaîne de compilation classique

code source

Analyse lexicale ↓

suite de mots

Analyse syntaxique ↓

abre de syntaxe abstraite (AST)

Analyses sémantiques ↓

AST typé

Simplification, optimisations ↓

langage intermédiaire

Émission de code ↓

code objet

Analyses lexicale et syntaxique

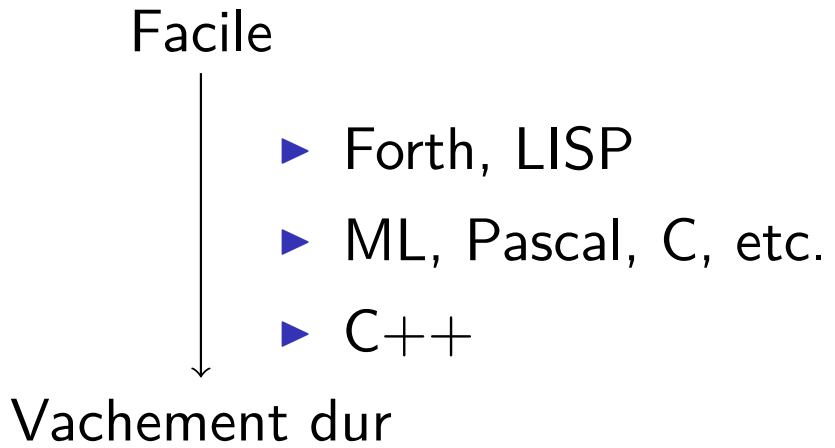
Syntaxe d'un langage

Les langages de programmation sont plus simples que les langues humaines, mais sont décrits de la même façon.

- Langage : ensemble des phrases possibles.
- Phrase : suite de mots correcte par rapport à une grammaire.
- Mot : élément d'un dictionnaire fini.

Classification des langages (1/4)

Suivant la complexité de la grammaire, il peut être plus ou moins difficile de vérifier qu'une phrase appartient au langage.



PLus une grammaire est difficile, plus

- la complexité (temps et espace) des algorithmes pour la traiter augmente,
- les **automates** permettant de les reconnaitre sont compliqués,
- le nombre de propriétés indécidables augmente,
- les messages d'erreur des **parseurs** sont illisibles.

Classification des langages (2/4)

Notation formelle d'une grammaire :

- T et N : **symboles terminaux** et **non terminaux**.
- R : Ensemble de **règles** : $\mathrm{Seq}(T \cup N) \rightarrow \mathrm{Seq}(T \cup N)$
  - lecture $\rightarrow$: production (énumération)
  - lecture $\leftarrow$: parsing (reconnaissance)
- S : un symbole de départ.

Exemple 1 :

- $T = \{a, b, c, d\}$, $N = \{S, X\}$
- $R = \left\{ \begin{array}{cccc} S \rightarrow aS, & S \rightarrow bS, & S \rightarrow cX, & S \rightarrow dX, \\ X \rightarrow cX, & X \rightarrow dX, & X \rightarrow \epsilon, & S \rightarrow \epsilon \end{array} \right\}$
- Productions valides :
  - $S \rightarrow \epsilon$,
  - $S \rightarrow aS \rightarrow aaS \rightarrow aacX \rightarrow aacdX \rightarrow aacd$,
  - $S \rightarrow aS \rightarrow adX \rightarrow ad$, etc.
- Langage : $[ab] * [cd] *$

Classification des langages (2/4)

Notation formelle d'une grammaire :

- T et N : **symboles terminaux** et **non terminaux**.
- R : Ensemble de **règles** : $\mathrm{Seq}(T \cup N) \rightarrow \mathrm{Seq}(T \cup N)$
  - lecture $\rightarrow$: production (énumération)
  - lecture $\leftarrow$: parsing (reconnaissance)
- S : un symbole de départ.

Exemple 2 :

- $T = \{a, b, c\}$, $N = \{S\}$
- $R = \{S \rightarrow c, S \rightarrow aSb\}$
- Productions valides :
  - $S \rightarrow c$,
  - $S \rightarrow aSb \rightarrow acb$,
  - $S \rightarrow aSb \rightarrow aaSbb \rightarrow aacbb$, etc.
- Langage : $a^n c b^n$

Classification des langages (2/4)

Notation formelle d'une grammaire :

- T et N : **symboles terminaux** et **non terminaux**.
- R : Ensemble de **règles** : $\mathrm{Seq}(T \cup N) \rightarrow \mathrm{Seq}(T \cup N)$
  - lecture $\rightarrow$: production (énumération)
  - lecture $\leftarrow$: parsing (reconnaissance)
- S : un symbole de départ.

Exemple 3 :

- $T = \{a, b, c\}$, $N = \{S\}$
- $R = \{S \rightarrow aSb, aSb \rightarrow aaSbb, aSb \rightarrow c\}$
- Productions valides :
  - $S \rightarrow aSb \rightarrow c$,
  - $S \rightarrow aSb \rightarrow aaSbb \rightarrow acb$,
  - $S \rightarrow aSb \rightarrow aaSbb \rightarrow aaaSbbb \rightarrow aacbb$, etc.
- Langage : $a^n c b^n$ (plus difficile à voir)

Classification des langages (3/4)

Hiérarchie de Chomsky :

Grammaires rationelles

$\subset$

Grammaires hors-contexte

$\subset$

Grammaires contextuelles

$\subset$

Grammaires générales

Classification des langages (3/4)

Hiérarchie de Chomsky :

Grammaires rationelles

Automate fini

$\subset$

Grammaires hors-contexte

Automate à pile

$\subset$

Grammaires contextuelles

Machine de turing à mémoire bornée

$\subset$

Grammaires générales

Machine de turing

Classification des langages (3/4)

Hiérarchie de Chomsky :

Grammaires rationelles

Automate fini

Règles de la forme : $N \to t,\ N \to tN$

$\subset$

Grammaires hors-contexte

Automate à pile

Règles de la forme : $N \to s, s \in \mathrm{Seq}(T \cup N)$

$\subset$

Grammaires contextuelles

Machine de turing à mémoire bornée

Règles de la forme : $s_1 N s_2 \to s_1 s s_2, (s, s_1, s_2) \in \mathrm{Seq}(T \cup N)^3$

$\subset$

Grammaires générales

Machine de turing

Règles de la forme : sans restriction

Classification des langages (4/4)

Notation courante : **BNF** (*Backus–Naur Form*)

- Règles de la forme `<non-term> ::=` *expression*
- Expressions : séquence (e_1 e_2), alternative $e_1 \mid e_2$
- Terminaux littéraux fixes : `"alors"`
- Ensembles de terminaux : `<integer>`

Exemple (opérations complétement parenthésées) :

```
<expr> ::= "(" <expr> <op> <expr> ")"
         | <integer>
         | <prim> "(" <args> ")"
         | "si" <expr> "alors" <expr> "sinon" <expr>
  <op> ::= "+" | "-" | "/" | "*"
<args> ::= <expr> "," <args>
         | <expr>
<prim> ::= "sqrt" | "cos" | "sin" | "tan"
```

Classification des langages : langages de programmation

Pour les langages de programmation, on utilise les deux classes les plus simples de grammaires

1. Analyse lexicale (*lexing*) : On définit la grammaire des **lexèmes** du langage, dont les **terminaux** sont les caractères.
   Assez simple $\Rightarrow$ grammaire rationnelle.
2. Analyse grammaticale (*parsing*) : On définit ensuite la grammaire des expressions, en utilisant les lexèmes déjà reconnus comme terminaux.
   Plus compliqué $\rightarrow$ grammaire hors-contexte.

Une fois la grammaire définie, on utilise un **générateur d'analyseur** qui vérifie la grammaire, et engendre un analyseur.

Analyse lexicale

Générateur d'analyseur lexical : `lex`, `ocamllex`, `ulex`, etc.
Prend en entrée :

- Un ensemble d'expressions rationnelles
- Pour chaque expression, du code appelé lorsque celle-ci a permis de reconnaitre un mot

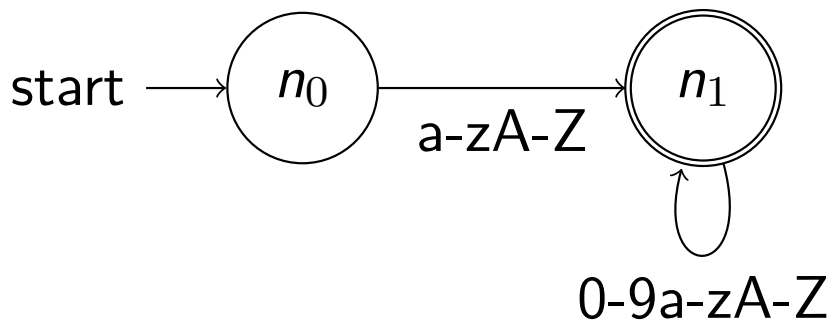
On fait la liaison avec l'analyse grammaticale grâce à :

- Un ensemble fini de lexèmes
- Les types de données associées le cas échéant
  ex: `BEGIN END INT<int> IDENT<string> ...`
- Le code de chaque expression devra produire un de ces lexèmes.

Exemples d'expressions (1/2)

Identifiants :

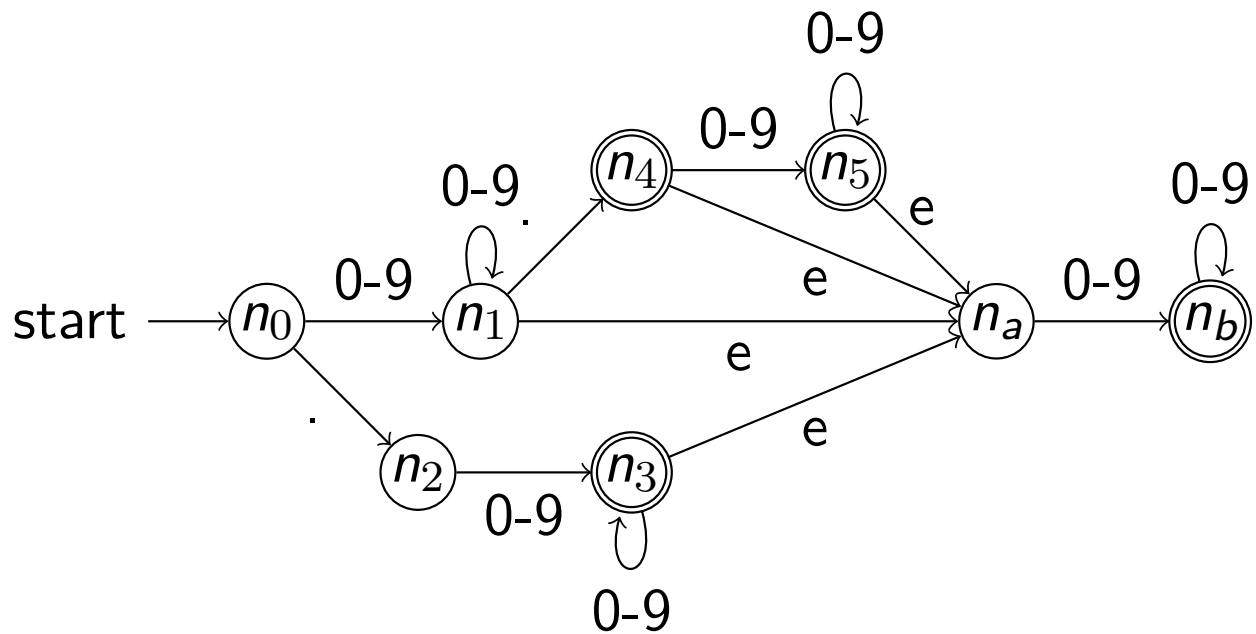
- Exemples : `azerty aZER_TY29 a44_`
- Expression : `[a-zA-Z][0-9a-zA-Z_]*`
- Automate :



Exemples d'expressions (2/2)

Flottants :

- Exemples : `.25 0. 0.22e17 3e8`
- Expression : en exercice
- Automate :



Exemple d'analyseur lexical pour C : `lex` / `flex`

Format du source

```
<DEF> <regexp>
%{ <code C apparaissant avant> %}

%%

<regexp> <action>

%%

<code C apparaissant après>
```

Cf. manuel de `flex`.

Exemple d'analyseur lexical pour C : lex / flex

Exemple

```
%{
#include <stdio.h>
%}
%%
[a-z]        printf("%c", ((yytext[0]-'a'+13)%26)+'a');
[\r\n]|(\r\n)  {printf("%s", yytext); fflush(stdout);}
.              printf("%c", yytext[0]);
%%
int main () {
  yyin = stdin;
  yylex();
}
```

compil: `flex rot13.lex && gcc lex.yy.c -lfl -o rot13`

Exemple d'analyseur lexical pour ocaml : ocamllex

Format du source

```
{
  <code OCaml exécuté avant>
}

let <nom> = <regexp>

rule <nom> = parse
    <regexp>              { <code associé> }
  | <regexp>              { <code associé> }
  | <regexp>              { <code associé> }
  | <regexp>              { <code associé> }

{
  <code OCaml exécuté après>
}
```

Exemple d'analyseur lexical pour ocaml : ocamllex

Utilisation autonome

```
{ open Printf open Char }

rule rot13 = parse
  | [ 'a'-'z' ] as c
    { printf "%c" (chr (((code c - code 'a' + 13) mod 26)
                        + code 'a')) }
  | [ '\n' '\r' ] | "\r\n" as s { printf "%s%!" s }
  | eof                         { raise Exit }
  | _ as c                      { printf "%c" c }
{ try
    let chan = Lexing.from_channel stdin in
    while true do
      rot13 chan
    done
  with Exit -> () }
```

```
ocamllex rot13.mll && ocamlopt rot13.ml -o rot13
```

Exemple d'analyseur lexical pour ocaml : ocamllex

Utilisation avec un parseur

```
{
open Parser
(* Parser définit
   type token = INT of int | OP of string
              | OPAR | CPAR *)
}

rule expr = parse
    [ ' ' '\t' ]              { expr lexbuf }
  | eof                       { END }
  | [ '0'-'9' ]+ as s         { INT (int_of_string s) }
  | [ '+' '-' '*' '/' ] as s  { OP s }
```

Analyse lexicale et conflits

Attention : les analyseurs lexicaux ne préviennent en général pas des conflits entre les règles.

```
{ open Printf }
rule expr = parse
  | [ 'a'-'z' ]+ as s          { printf "lu %s\n%!" s }
  | "toto"                  { printf "ne doit pas arriver" }
```

Analyse grammaticale

Générateur d'analyseur lexical : ex. `yacc`, `menhir`, `antlr`, `javacc`

Deux types principaux, correspondant à deux restrictions des grammaires algébriques (hors-contexte) :

1. LL(k) : calcule la **dérivation gauche**
   (ré-écriture du non terminal le plus à gauche)
2. LR(k) : calcule la **dérivation droite**
   (ré-écriture du non terminal le plus à droite)

Le k donne le nombre de lexème que l'analyseur doit tester pour prendre chaque décision.

Ambiguïtés

En général, on veut une grammaire **non-ambigüe** : une phrase correspondant à un seul arbre de dérivation.

- Différents analyseurs $\rightarrow$ même arbre
- Mieux acceptées par les générateurs d'analyseurs

Il faut souvent réécrire la grammaire :

- Formatage dépendant du générateur d'analyseur
- La grammaire ne représente plus la structure
- Perte en modularité (explicitation manuelle des cas)

Analyseur LL(k)

LL(1) : Approche utilisée pour écrire un parseur à la main, sous forme de fonctions mutuellement récursives.

- On analyse le flot lexème par lexème,
- il suffit de regarder un lexème pour choisir la fonction à appeler,
- on peut plus facilement donner des messages intelligibles,
- il faut par contre **vérifier sa grammaire** avant l'implantation.

Restrictions :

- Récursion gauche interdite (boucle infinie)
- Pas d'expansions commençant par le même symbole (choix) $\rightarrow$ factorisation du début dans une règle intermédiaire

NB: Il n'est pas possible d'analyser les expressions arithmétiques en LL(1).

Analyseur LL(k)

Exemple de réécriture LL(1) d'une grammaire :

- Récursion gauche $\{E \to E + E,\ E \to x\}$
  donne $\{E \to xZ,\ Z \to +xZ,\ Z \to \epsilon\}$
- Factorisation du début $\{E \to AB,\ E \to AC\}$
  donne $\longrightarrow \{E \to AX,\ X \to B,\ X \to C\}$
- Ambiguîté terminal/non terminal $\{E \to Za,\ Z \to a,\ Z \to \epsilon\}$
  donne $\longrightarrow \{E \to a,\ E \to aa\}$

Analyseur LR(k)

LR(1) : Approche utilisée par la plupart des générateurs d'analyseurs.

- Grammaires plus souples LL(1) $\subset$ LR(1),
- détection des ambigüités et les conflits,
- messages d'erreurs plus difficiles à implanter,
- nécessite en général une machinerie.

Restrictions : conflits SHIFT/REDUCE (cf. suite)

Analyseur LR(1) : fonctionnement

On empile les lexèmes (SHIFT), et on décide quand on reconnait une règle au sommet, et on réécrit le sommet de pile (REDUCE).

Exemple : $\{S \to (E),\ E \to E;X,\ E \to X,\ X \to a,\ X \to b,\ \}$

Pile	Flux	
	(a ; b ; a)	S
(	a ; b ; a)	S
(a	; b ; a)	R
(X	; b ; a)	R
(E	; b ; a)	S
(E ;	b ; a)	S
(E ; b	; a)	R
(E ; X	; a)	R
(E	; a)	S
(E ;	a)	S
(E ; a	)	R
(E ; X	)	R
(E	)	S
(E)		R
S		-

Analyseur LR(1) : conflits

- REDUCE/REDUCE : l'analyseur pourrait réduire le sommet de pile en deux non terminaux différent.
  Solutions :
  - Erreur/ambigüité : grammaire à revoir
  - Trop peu d'avance : factoriser
  - Ambigüité simple : règles de l'analyseur
- SHIFT/REDUCE : l'analyseur pourrait réduire le sommet de pile ou continuer à empiler.
  Solutions :
  - factoriser le début, comme pour LL(1)
  - utiliser les règles d'assoc. et de distrib. du générateur

Les générateurs d'analyseurs sont plus ou moins loquaces sur les erreurs de grammaires, ne pas hésiter à en changer si une grammaire est difficile à reformater.

Analyseur LR(1) : menhir

Format du source

```
(* déclaration des tokens *)
%token<type> <TOK1> ...
(* déclaration des règles typées *)
%start<type> <règle1> ...
(* triche *)
%<nonassoc|left|right> <TOK> ... (* faible prio *)
%<nonassoc|left|right> <TOK> ... (* haute prio *)

%%
règle:
  | TOK ; TOK ; TOK { <code>}
  | TOK ; n = TOK ; TOK { <code utilisant n>}
  | TOK ; n = règle ; TOK { <code utilisant n>}
%%

<code>
```

Analyseur LR(1) : menhir

exemple

```
%token<int> INT
%token PLUS TIMES
%start<int> expr

%%

expr:
| i = INT { i }
| e = expr ; PLUS; f = expr    { e + f }
| e = expr ; TIMES; f = expr   { e * f }
```

Analyseur LR(1) : menhir

Réécriture LR possible

```
%token<int> INT
%token PLUS TIMES
%start<int> expr

%%

expr:
| e = sexpr                          { e }
| e = sexpr ; TIMES; f = expr        { e * f }
sexpr:
| i = INT                            { i }
| e = INT ; PLUS; f = sexpr          { e + f }
```

Analyseur LR(1) : menhir

Sans réécriture, avec les règles de priorité

```
%token<int> INT
%token PLUS TIMES
%left PLUS
%left TIMES
%start<int> expr

%%

expr:
| i = INT { i }
| e = expr ; PLUS; f = expr    { e + f }
| e = expr ; TIMES; f = expr   { e * f }
```

Retour sur les règles de priorité

- %left : on réduit avant d'empiler
  $E + E + E \rightarrow (E + E) + E$
- %right : empile avant de réduire
  $E + E + E \rightarrow E + (E + E)$
- priorité inférieure (apparaît avant), on continue d'empiler
  $\text{prio}(+) < \text{prio}(*)$: $E + E * ... \rightarrow E + E * ...$
- priorité supérieure (apparaît après), on réduit
  $\text{prio}(*) > \text{prio}(+)$: $E * E + ... \rightarrow (E * E) + ...$

Références

- Pages WikiPédia **en anglais** sur les langages
  (celles sur les analyseurs sont moins bonnes)
- Livre *Compilers: Principles, Techniques, and Tools*
- chapitre 11 de DAOC
  `http://www.pps.univ-paris-diderot.fr/Livres/ora/DA-OCAML`
- Manuel de `menhir` :
  `http://gallium.inria.fr/~fpottier/menhir/`
- Manuel de `flex` :
  `http://flex.sourceforge.net/manual/`