Rappels et compléments de compilation
Analyses lexicale et syntaxique
Cours de Compilation Avancée (MU4IN504)

Benjamin Canou & Emmanuel Chailloux
Sorbonne Université

Année 2019/2020 – Semaine 1

Description de l'UE

Des machines virtuelles aux machines réelles

- langage intermédiaire, machine virtuelle,
- structures de contrôle de haut niveau
- implantation bibliothèque d'exécution (gestion mémoire)
- passage au code natif, construction du graphe de contrôle
- analyse et optimisation du code engendré
- optimisation pour la hiérarchie mémoire

Plan du cours

Première partie : Cours par **Emmanuel Chailloux**, TD/TME par **Darius Mercadier**

- Cours 1 : Rappels, analyseurs
- Cours 2 et 3 : Machines virtuelles et bibliothèques d'exécution
- Cours 4 : Modèles mémoire
- Cours 5 : Contrôle de haut niveau : exceptions, continuations, concurrence

Deuxième partie : Cours et TD/TME par **Karine Heydemann**

- Cours 6 : Analyse du flot de contrôle
- Cours 7 : Ordonnancement de code
- Cours 8 et 9 : Elimination des redondances, analyse du flot de données
- Cours 10 : Allocation de registres

Informations pratiques

Sites

https ://www-master.ufr-info-p6.jussieu.fr/2020/ca
http ://www-apr.lip6.fr/~chaillou/Public/enseignement
http ://www-soc.lip6.fr/~heydeman/

Evaluation

- 1ère session
  - un devoir par partie (20% + 20%)
  - un examen papier pour 60% (date probable : 19 mai 2020)
- 2ème session
  - un examen de rattrapage pour 100 % (à partir du 15 juin)

Questions préalables

Comprendre comment faire exécuter un programme écrit dans un langage informatique donné ?

et pour cela on tachera de répondre aux questions suivantes :

- qu'est-ce qu'un programme ?
- qu'appelle-t-on langage de haut-niveau, de bas niveau ?
- qu'est-ce qu'un interprète/compilateur ?
- quelles sont les principales phases d'un compilateur ?
- qu'est-ce qu'un exécutable ?
- qu'est-ce qu'une bibliothèque d'exécution ?
- qu'est-ce qu'une machine virtuelle ?
- qu'est-ce que la gestion automatique de mémoire ?

Qu'est-ce qu'un programme ?

un texte ayant un sens (décrivant un algorithme) :

- suite de caractères formant des mots
- permettant de construire des expressions et/ou des instructions
- en suivant une syntaxe précise donnée dans une grammaire
- et vérifiant certaines propriétés (typage, ...)

que l'on exécute ensuite :

- soit directement en évaluant l'arbre de syntaxe abstraite produit par un **interpréteur**
- soit en le traduisant par un **compilateur** vers le code d'une machine (réelle ou virtuelle), qui une fois relié avec les bibliothèques d'exécution, produire un code exécutable par la machine.

pour produire un résultat.

qu'appelle-t-on langage de bas-niveau, haut-niveau (1) ?

dans le monde séquentiel :

- bas-niveau :
  langages proches de la machine : modèle impératif
  (assembleur, C, ...)
  - un contrôle simple :
    avec des structures de contrôle simples : séquence, alternative (if then else), itérative (boucle), appel de procédures ou fonctions globales
  - avec une gestion de la mémoire à la main (manipulation de pointeurs)

qu'appelle-t-on langage de bas-niveau, haut-niveau (2) ?

- haut-niveau
- du contrôle de plus haut niveau
  - exceptions, continuations
- et d'autres modèles de calcul :
  - fonctionnel : Lisp, JavaScript, OCaml, F#, Swift, Haskell
  - logique : Prolog
  - objet : Java, C#, Obj-C, C++, JavaScript, Python, OCaml, Swift, F#
- y compris concurrents : threads (coopératif, préemptif), canaux synchrones, prog synchrone, futures, streams, ...

Qu'est-ce qu'un interprète/compilateur ?

La syntaxe d'un programme a été vérifiée par un autre programme. Celui-ci construit en général un arbre de syntaxe abstraite des expressions et instructions du programme.

- un **interprète** va évaluer les nœuds de cet arbre pour produire un résultat
- un **compilateur** lui va traduire cet arbre dans une suite d'instructions d'une machine réelle (processeur x86) ou virtuelle (JVM : machine virtuelle Java).

Comment faire exécuter les instructions d'une machine virtuelle : par un programme qui simule les instructions de cette machine (la commande `java` ou la commande `ocamlrun`).

Qu'est-ce qu'un exécutable, une bibliothèque d'exécution ?

Un programme exécutable sur un système d'une machine nécessite de relier le code traduit par un compilateur avec

- certaines bibliothèques du système (I/O)
- des fonctionalités de la bibliothèque d'exécution fournie par le langage (gestion mémoire, mécanisme d'exceptions, ...)

Dans le cas d'utilisation d'une machine virtuelle, l'interpréteur de la machine virtuelle est lui-même un programme exécutable qui va interpréter les instructions de la machine virtuelle.

Qu'est-ce qu'une machine virtuelle ?

Une machine virtuelle définit :

- des instructions de calcul (primitives), de sauts dans le code, d'accès aux environnement, de manipulation de la pile, d'allocation mémoire dans le tas et d'appel aux fonctions et primitives
- des zones mémoires spécifiques : code, environnement global, pile et tas
- une bibliothèque d'exécution principalement pour la gestion mémoire, et souvent la gestion des exceptions

Historiquement, on parle de byte-code (code-octet) pour ces instructions.

Qu'est-ce que la gestion automatique de mémoire

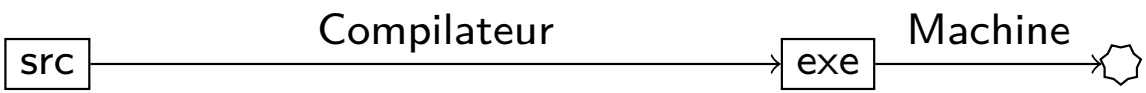
- en C : l'allocation est explicite (`malloc`) et la récupération mémoire aussi (`free`)
- en Java : l'allocation est explicite (**new**) mais la récupération est implicite :
  - il n'y a pas d'instructions pour l'indiquer mais plusieurs techniques possibles (compteurs de référence, algorithme explorateurs, ...)

Rappels

Qu'est-ce que la compilation ?

Principe de base de la compilation :

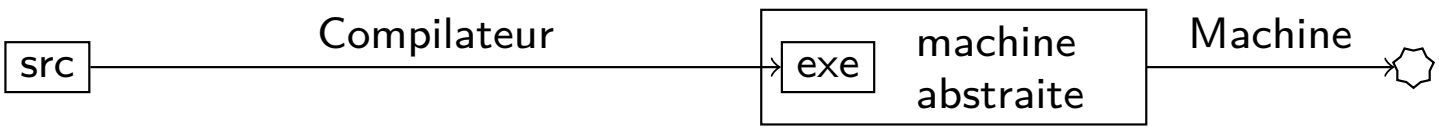
Traduction du **code source** vers du **code machine** (*code natif*).



Qu'est-ce que la compilation ?

Compilation pour une machine virtuelle (*VM*) :

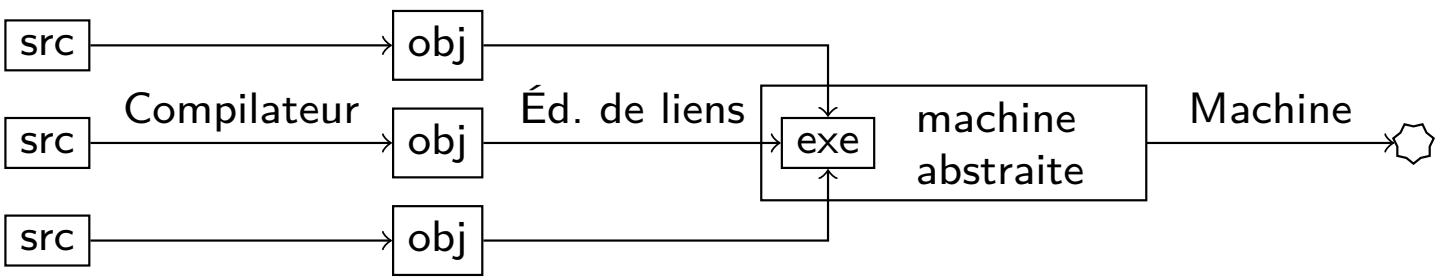
Production de **code-octet *(bytecode)*** interprété ou compilé à la volée vers du code machine.



Qu'est-ce que la compilation ?

Compilation séparée :

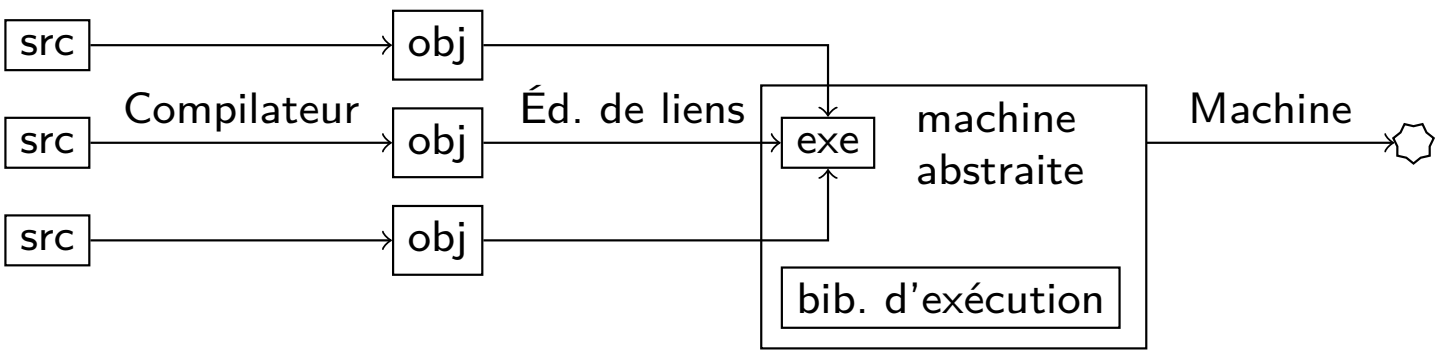
Liaison de **fichiers objet**, un fichier objet par **unité de compilation** du langage (*classe*, *module*, *package*, *etc.*).



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 méthodes, continuations, etc.*).

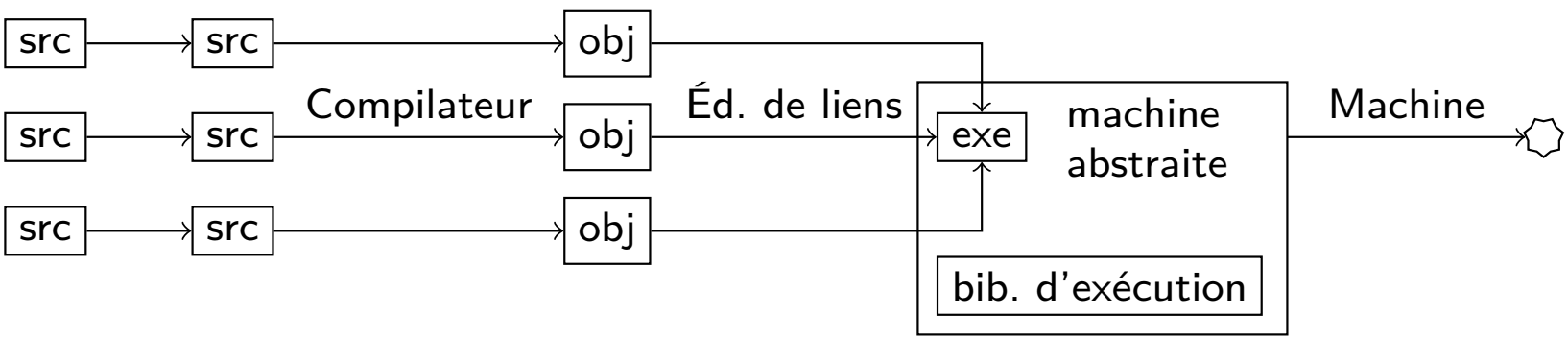
- VM : intégré dans la machine virtuelle
- Code natif : lié dans l'exécutable



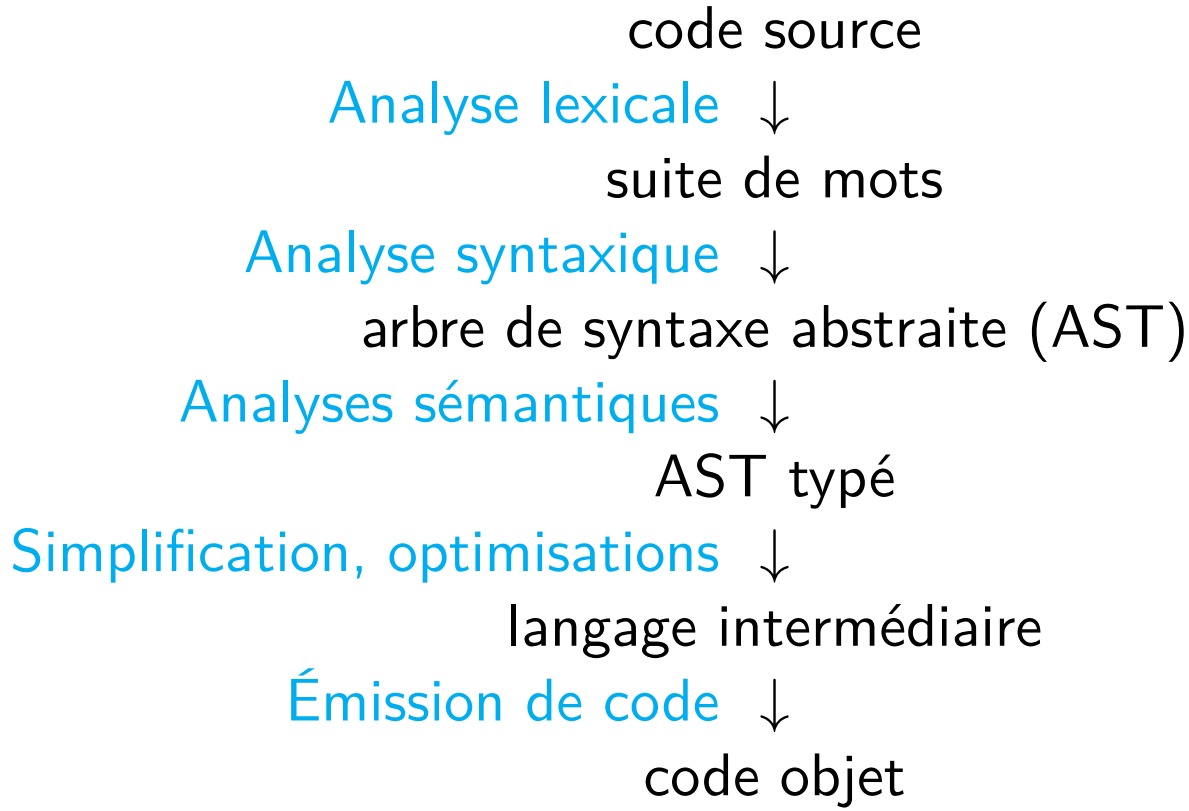
Qu'est-ce que la compilation ?

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



Analyses lexicale et syntaxique

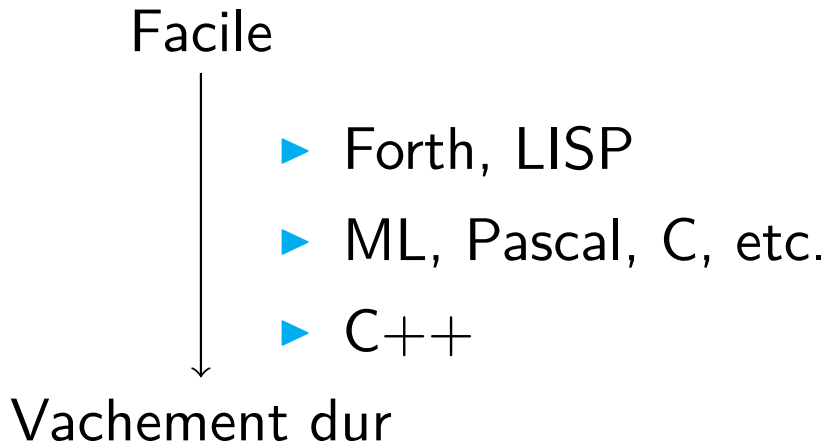
Syntaxe d'un langage

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 rationnelles

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`, `jlex`, etc.
Prend en entrée :

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

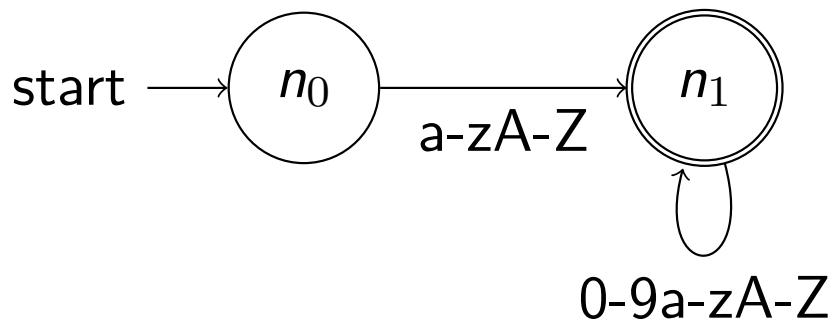
On fait la liaison avec l'analyse grammaticale grâce à :

- 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 :

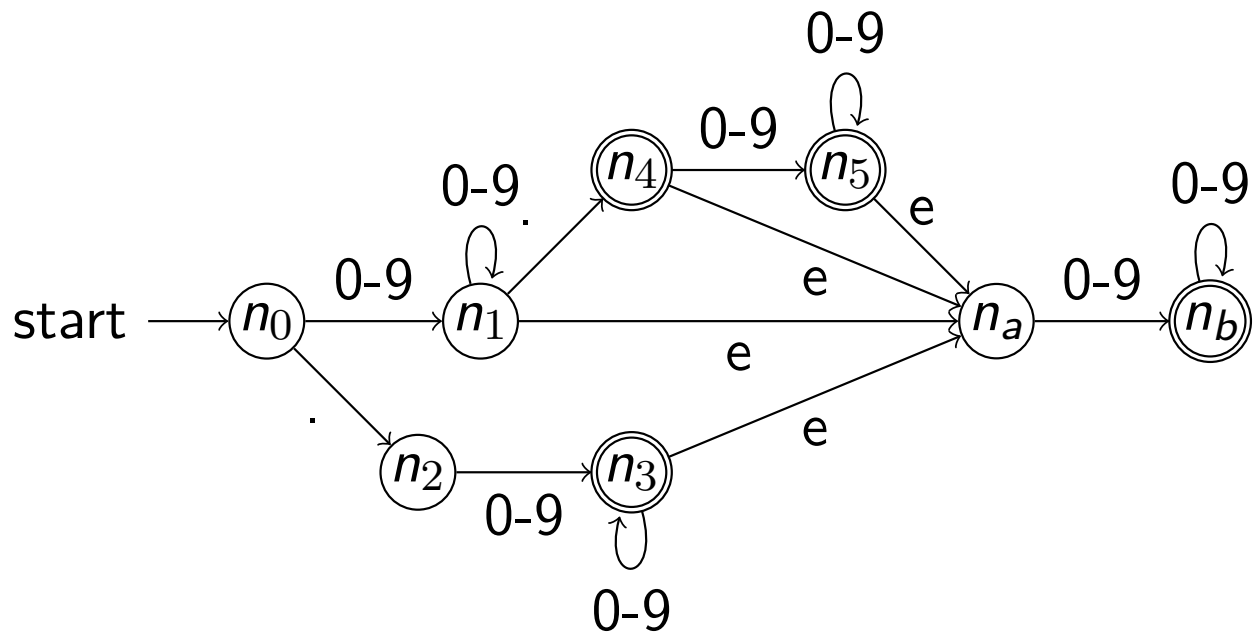
- Exemples : `azerty aZER_TY29 a44_`
- Expression : `[a-zA-Z][0-9a-zA-Z_]*`
- Automate :



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 }
```

Exemple d'analyseur lexical pour Java :

JFlex* (The Fast Lexical Analyser Generator for Java)
à installer : `http://www.jflex.de/`

Format général d'un fichier d'entrée pour JFlex : (de celui de Lex)

```
Code utilisateur
%%
Options et  déclarations
%%
Règles lexicales
```

Exemples du cours

```
import java.lang.* ;
%%
%class Exemple
%public
%standalone
%{
 public static char add(char a) {
   int ai = (int)a;
   int ri = (((ai-(int)'a'+13)%26)+ (int)'a');
   return  (char)ri ;
 }
%}
%%
[a-z]       { System.out.print(add(yytext().charAt(0))) ; }
[\r\n]|(\r\n)  { System.out.print(yytext()) ; }
.           { System.out.print(yytext().charAt(0)) ; }
```

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 syntaxique : 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-ambiguë** : 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)

Langages reconnus et ambiguité (1)

Une grammaire G définit un langage $L(G)$ sur l'alphabet Σ dont les éléments sont les mots engendrés par dérivation à partir du symbole S de départ :
$L(G) := \{\omega \in \Sigma * | S \rightarrow^+ \omega\}$

Diiférentes grammaires pour les expressions arithmériques

- exemple 1 : G_{ARITH1}
  - $T = \{int, (,), +, -, *, /\}$, $N = \{E\}$
  - $R = \left\{ \begin{array}{ccc} E \rightarrow int, & E \rightarrow (E), & E \rightarrow E + E, \\ E \rightarrow E - E, & E \rightarrow E * E, & E \rightarrow E/E \end{array} \right\}$
  - symbole de départ E
- l'expression : $3 + 4 * 5$ peut être produite de deux manières :
  - $E \rightarrow E + E \rightarrow 3 + E \rightarrow 3 + E * E \rightarrow 3 + 4 * E \rightarrow 3 + 4 * 5$
  - ou $E \rightarrow E * E \rightarrow E * 5 \rightarrow E + E * 5 \rightarrow E + 4 * 5 \rightarrow 3 + 4 * 5$

On dit qu'une grammaire est ambiguë, s'il existe au moins une phrase de son langage reconnue par deux dérivations construisant des arbres différents.

Langages reconnus et ambiguité (2)

- exemple 2 : G_{ARITH2}
  - $T = \{int, (,), +, -, *, /\}$, $N = \{E, T, F\}$
  - $R = \left\{ \begin{array}{ccc} E \rightarrow E+T, & E \rightarrow E-T, & E \rightarrow T, T \rightarrow T*F, \\ T \rightarrow T/F, & T \rightarrow F, & F \rightarrow (E), F \rightarrow int \end{array} \right\}$
  - symbole de départ E
  - l'expression : $3+4*5$ est reconnue en construisant le même arbre de dérivation :
    - $E \rightarrow E+T \rightarrow T+T \rightarrow F+T \rightarrow 3+T \rightarrow 3+T*F \rightarrow 3+F*F \rightarrow 3+4*T \rightarrow 3+4*5$

la grammaire n'est plus ambigue, mais certains termes reconnus par G_{ARITH1} ne le sont plus ici comme :
$3+4*5+6$
il convient alors de parenthéser :
$(3+4)*(5+6)$ ou $3+((4*5)+6)$
pour pouvoir reconnaitre une telle expression arithmétique.

Langages reconnus et ambiguité (3)

- exemple 3 : G_{ARITH3}
  - $T = \{int, (,), +, -, *, /\}$, $N = \{E, E', T, T', F\}$
  - $R =$
    $$\left\{ \begin{array}{cccc} E \rightarrow TE', & E' \rightarrow +TE', & E' \rightarrow -TE', & E' \rightarrow \epsilon, \quad T \rightarrow FT' \\ T' \rightarrow *FT', & T' \rightarrow /FT', & F \rightarrow (E), & F \rightarrow int \end{array} \right\}$$
  - symbole de départ E
  - l'expression : $3 + 4 * 5$ ne peut produire qu'un seul arbre de dérivation :
    - $E \rightarrow TE' \rightarrow FT'E' \rightarrow FE' \rightarrow 3E' \rightarrow 3 + TE' \rightarrow 3 + FT'E' \rightarrow 3 + 4T'E' \rightarrow 3 + 4 * FT'E' \rightarrow 3 + 4 * 5T'E' \rightarrow 3 + 4 * 5E' \rightarrow 3 + 4 * 5$

la grammaire n'est plus ambigue, et le premier symbole en lecture de gauche à droite est unique, mais certains termes reconnus par G_{ARITH2}ne le sont plus. Là aussi il faut parenthéser.

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) selon les grammaires G_{ARITH1} et G_{ARITH2}.

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 $\longrightarrow \{E \to AX,\ X \to B,\ X \to C\}$
- Ambiguïté 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 ambiguïtés 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 différents.
  Solutions :
  - Erreur/ambiguïté : grammaire à revoir
  - Trop peu d'avance : factoriser
  - Ambiguïté 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
  $\text{prio}(+) < \text{prio}(*) : E + E * \ldots \rightarrow E + E * \ldots$
- priorité supérieure (apparaît après), on réduit
  $\text{prio}(*) > \text{prio}(+) : E * E + \ldots \rightarrow (E * E) + \ldots$

Cup : un générateur d'analyseurs pour Java

- Java-Based Constructor of Useful Parsers (CUP)

  Permet de construire des analyseurs LALR, sous-ensemble des analyseurs LR mais avec une optimisation qui effectue des unions dans les tables (cela peut engendrer des conflits mais réduit fortement le nombre d'entrée dans la table).

- un fichier de description (extension `.cup`)
- principaux éléments engendrés :
  - classe `parser` : contient l'analyseur
  - classe `sym` : contient la définition de chaque terminal (associé à une constante) (codé par un entier)

CUP : fichier de description

- inclusion de code :
  - paquetages et importations, `init` pour l'initialisation, `scan` pour la lecture d'un symbole (par défaut `next.token`), parser et action pour ajouter du code dans la classe `Parser`.
- liste des terminaux et non terminaux
- déclaration de priorité et d'associativité
- grammaire en tant que telle

Exemple : un calculateur (1)

- 2 fichiers :
  - calc.cup : définit la grammaire et les actions associées aux règles
  - `scanner.java` : qui fournit un analyseur lexical à la main (classe scanner) et une classe (`Main`) avec un point d'entrée

```
public class scanner { /* ... */
 private SymbolFactory sf = new DefaultSymbolFactory();
 public Symbol next_token() {
  for (;;)
   switch (next_char){
    case ';': advance(); return sf.newSymbol("SEMI",sym.SEMI);
    case '+': advance(); return sf.newSymbol("PLUS",sym.PLUS);
    case '-': advance(); return sf.newSymbol("MINUS",sym.MINUS);
    case '*': advance(); return sf.newSymbol("TIMES",sym.TIMES);
    case '(': advance(); return sf.newSymbol("LPAREN",sym.LPAREN);
    case ')': advance(); return sf.newSymbol("RPAREN",sym.RPAREN);
/* ... */
  }}
```

Exemple : code à inclure (2)

un exemple d'un calculateur (voir doc cup) :

```
import java_cup.runtime.*;

parser code {:
    scanner s;
    Parser(scanner s){
        this.s=s;
    }
    :}
init with {: s.init(); :};
scan with {: return s.next_token(); :};
```

Exemple : déclaration des symboles (3)

```
non terminal              expr_list;
non terminal Integer      expr;
/* Precedences */
precedence left PLUS, MINUS;
precedence left TIMES;
precedence left UMINUS;
```

Exemple : description de la grammaire (4)

calcul direct mise à jour de RESULT :

```
expr_list ::= expr_list expr:e SEMI      {: System.out.println("--> "+e);:}
    | expr:e SEMI                       {: System.out.println("==> "+e);:}
;
expr      ::= expr:e1 PLUS  expr:e2          {: RESULT = e1+e2;        :}
            | expr:e1 MINUS expr:e2          {: RESULT = e1-e2;        :}
            | expr:e1 TIMES expr:e2          {: RESULT = e1*e2;        :}
            | MINUS expr:e                   {: RESULT = -e;           :}
            %prec UMINUS
            | LPAREN expr:e RPAREN           {: RESULT = e;            :}
            | NUMBER:n                       {: RESULT = n;            :}
            ;
```

Compilation et exécution

- compilation :

```
java -jar ../java-cup-11b.jar -interface -parser Parser calc.cup
javac -cp ../java-cup-11b-runtime.jar:. *.java
```

- exécution :

```
echo "1 + 3; 8 * 9 ; 33 - 1;" | java -cp ../java-cup-11b-runtime.jar:. Main
==> 4
--> 72
--> 32
```

Grammaires contextuelles (1)

La poursuite de l'analyse d'un flot de lexèmes ne dépend pas des valeurs syntaxiques déjà traitées. Ce n'est pas le cas du langage L décrit par la formule suivante :

$L ::= wCw$ où $w \in (A|B]^*$

où A, B et C sont des symboles terminaux. On a écrit wCw (avec $w \in (A|B)^*$ et non simplement $(A|B)^* w (A|B)^*$ car on voulait avoir le même mot à gauche et à droite du C médian.

exemple avec les parsers en OCaml.

Grammaires contextuelles (2)

```
# type token = A | B | C ;;
# let rec parse_w1 s =
    parser
   |  [<'A; l = parse_w1  >] -> (parser [<'A >] -> "a")::l
   | [<'B; l = parse_w1  >] -> (parser [<'B >] -> "b")::l
   | [< >] -> [] ;;
val parse_w1 : token Stream.t -> (token Stream.t -> string) list = <fun>

# let rec parse_w2 l =
   match l with
     p::pl -> (parser [< x = p; l = (parse_w2 pl)  >] -> x^l)
   | [] -> parser [<>] -> "" ;;
val parse_w2 : ('a Stream.t -> string) list -> 'a Stream.t -> string = <fun>

# let parse_L = parser  [< l = parse_w1 ; 'C; r = (parse_w2 l) >] -> r ;;
val parse_L : token Stream.t -> string = <fun>

# parse_L [< 'A; 'B; 'B; 'C; 'A; 'B; 'B >];;
- : string = "abb"

# parse_L [< 'A; 'B; 'C; 'B; 'A >];;
Uncaught exception: Stream.Error("")
```

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://caml.inria.fr/pub/docs/oreilly-book/html/index.html`
  `https://www-apr.lip6.fr/~chaillou/Public/DA-OCAML/`
- Manuel de `menhir` :
  `http://gallium.inria.fr/~fpottier/menhir/`
- Manuel de `flex` :
  `http://flex.sourceforge.net/manual/`
- jflex : `http://www.jflex.de`
- cup : `http://www2.cs.tum.edu/projects/cup/`