UPMC Paris Universitas
Master Informatique - STL

Compilation Avancée (MI190) 2014
Cours 1 : Introduction & Analyseurs

Carlos Agon
agonc@ircam.fr

C'est quoi la compilation ?

Traduire : un programme vers du code en langage machine

- 50's assembler
  du code machine textuel pour des langages de haut niveau (Fortran)

- 60's Les langages évoluent (la récursion) le langage machine suit (utilisation d'une pile), mais pas tant que ça.

- 80's Représentation automatique des données, le langage machine ne suit plus..

Il y a de plus en plus un écart entre l'expressivité des langages des haut niveau et celle du langage machine. C'est grave ?

… le compilateur doit s'occuper de cette fracture.

Autres tâches

Link : liaison des fichiers des unités de compilation (module, class, interface, package, etc.)

Runtime : utilisation des bibliothèques d'exécution (i/o, gc, appels de méthodes, etc.)

VM : production de *bytecode* et interprétation/compilation (jit) vers du code machine

Transformations source-à-source : macros – interopérabilité (C)

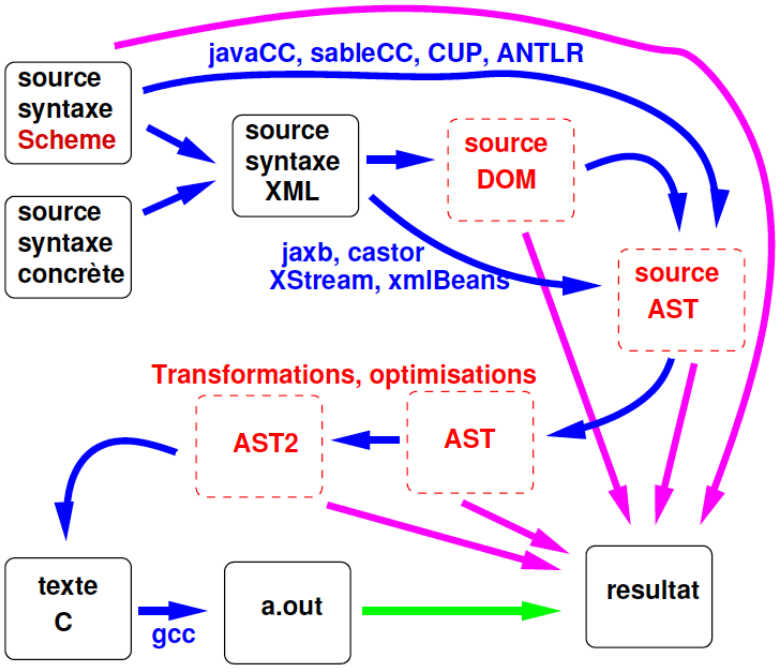
Optimisations : temps, espace, énergie.

Sûreté : typage.

…

… la compilation est une chaîne de processus.

Chaîne de compilation (Souvenir d'ILP)



Chaîne de compilation classique

Code source

Analyse lexicale

Analyse syntaxique

Analyses statiques

Génération

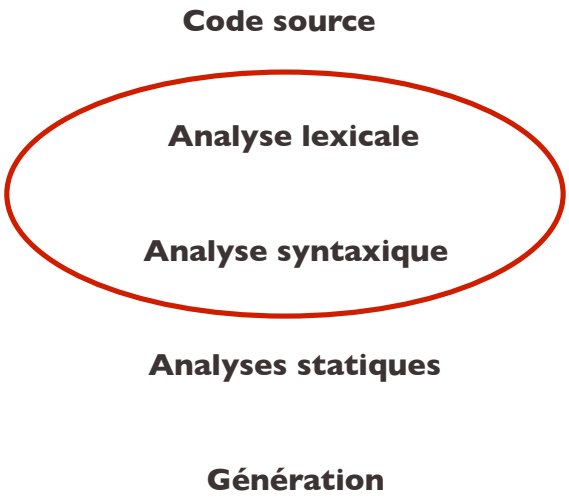
Informations

Première partie
Cours et TD/TME par Carlos Agon
- Cours 1 : Introduction, analyseurs
- Cours 2-3 : Machines abstraites, virtuelles et bibliothèques d'exécution
- Cours 4 : Modèles mémoire - Garbage collection
- Cours 5 : Contrôle de haut niveau

Deuxième partie
Cours et TD/TME par Karine Heydemann
- Mémoire cache et optimisation pour la hiérarchie mémoire
- Architecture matérielle basée sur le parallélisme d'instructions
- Ordonnancement et optimisation

Evaluation
- Un examen 60 %
- Un projet par partie (20 % + 20 %)

Cours 1



Langages formels

Définitions

Symbole : signe primitif (lettre, chiffre, point, ligne, …)

Alphabet : ensemble fini de symboles (Σ)

Chaîne : séquence finie de symboles (la chaîne vide ε)

Langages formel : ensemble de chaînes définies sur un alphabet Σ

Problèmes :

- Comment définir un langage en intension ?
- Comment calculer si une chaîne ω appartient ou non à L ?

Hiérarchie des langages

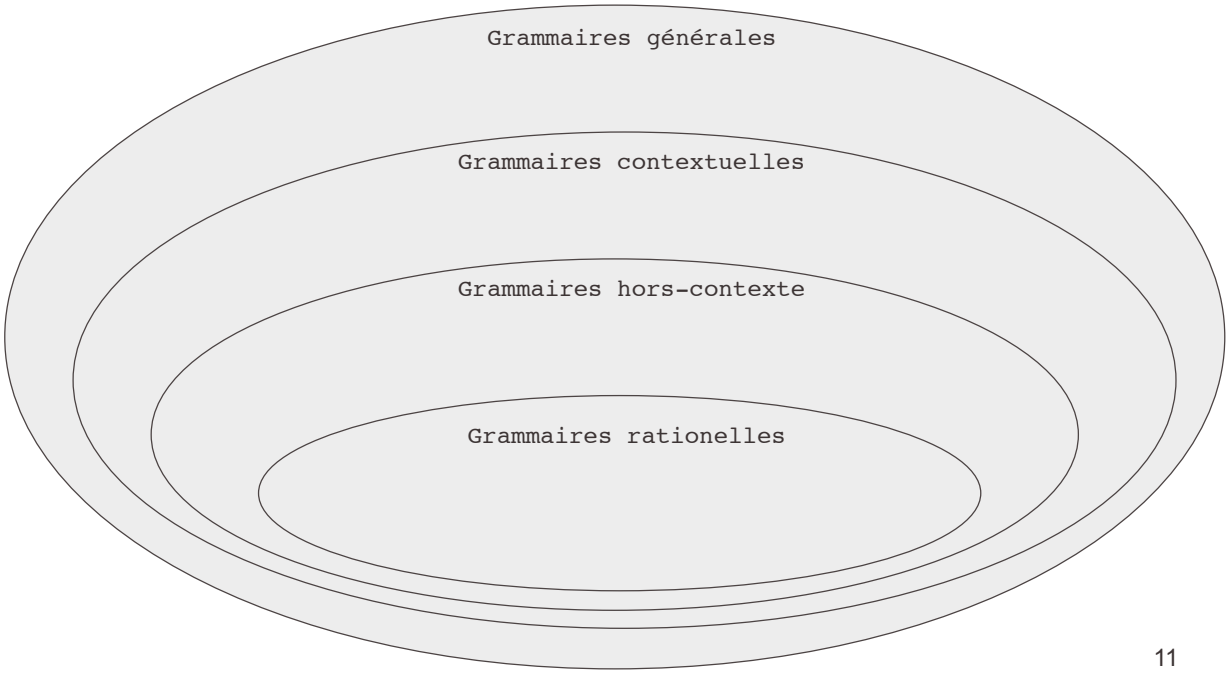
Caractérisation des langages en fonction de la difficulté (calculabilité, espace, temps) de reconnaissance d'une chaîne.

Langages finis : plutôt facile.

Langages infinis: hiérarchie de Chomsky :

- rationnels ou réguliers,
- indépendants du contexte,
- dépendants et
- récursivement énumérables.

Hiérarchie de Chomsky



Grammaires formelles

T : symboles terminaux
N : symboles non terminaux
R : $\{\alpha \rightarrow \beta\}$ avec $\alpha \in (\mathbf{T} \cup \mathbf{N})^*$ et $\beta \in (\mathbf{T} \cup \mathbf{N})^*$
S : un symbole de départ e **N**

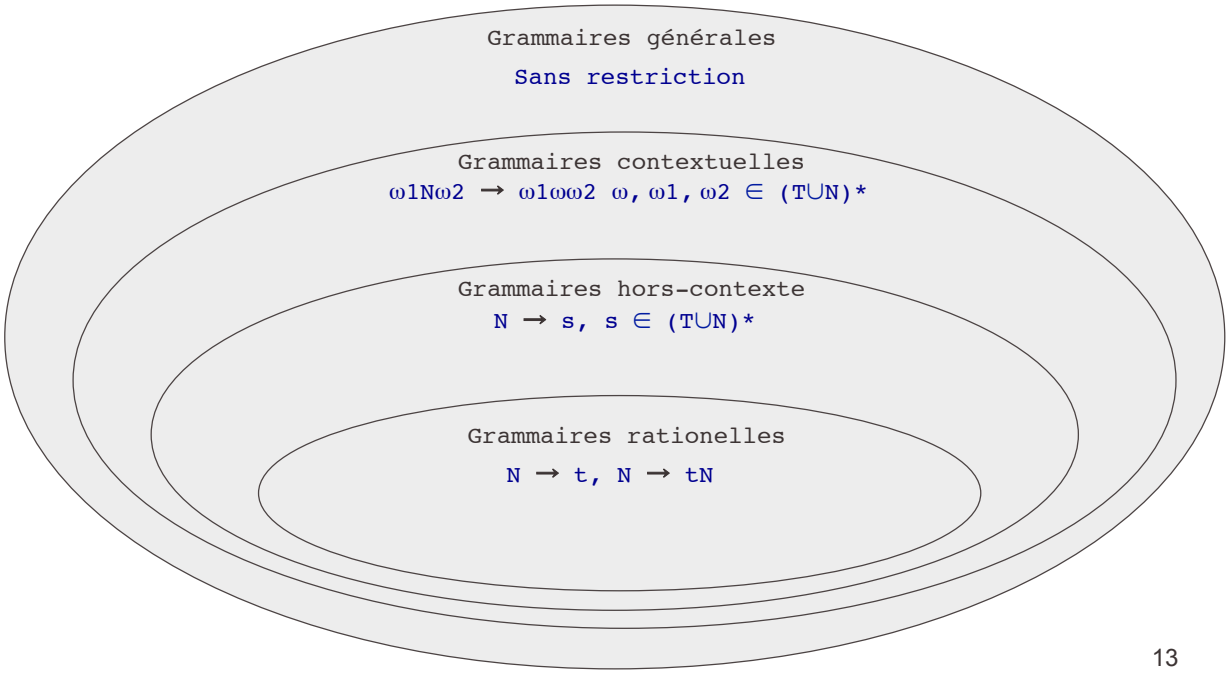
Exemple :

- **T** = {a,b,c}
- **N** = {S}
- **S** = S
- **R** = S → aSb
  aSb → aaSbb
  aSb → c

Acceptées : c, aacbb, acb,…
Non-acceptées : acab,…

Langage : $a^n c b^n$

Hiérarchie de Chomsky



Langages réguliers

Soit $\Sigma = \{a1,\ldots,an\}$
Les langages triviaux : $Li = \{ai\}$

Et trois opérations :

union : $L \cup L' = \{\omega \mid \omega \in L \text{ ou } \omega \in L'\}$
produit : $L.L' = \{\upsilon.\omega \mid \upsilon \in L \text{ et } \omega \in L'\}$

Itération : $L^* = \bigcup_{i \geq 0} L^i$

$L^0 = \{\varepsilon\}$; $L^1 = L$; $L^2 = L.L^1$; … ; $L^{i+1} = L.L^i$

Langage régulier : $L \subseteq \Sigma^*$
ssi L peut être obtenu à partir des langages triviaux sur Σ uniquement par l'application des opérations d'union, de produit et d'itération.

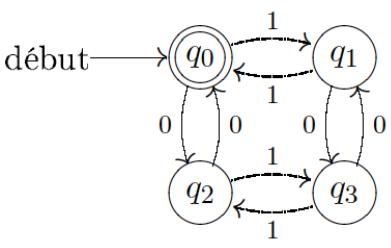
automates

Comment calculer si une chaîne ω appartient ou non à L ?

Automate fini déterministe (AFD) : $(Q, \Sigma, \delta, q_0, F)$ tq.

- $Q = \{ q_0, \ldots, q_k \}$
- Σ = alphabet $\{x_1, x_2, \ldots, x_n\}$
- $\delta : Q \times \Sigma \rightarrow Q$
- $q_0 \in Q$ état initial
- $F \subseteq Q$ états finaux

AFD exemple



États	Entrées 0	1
q_0	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

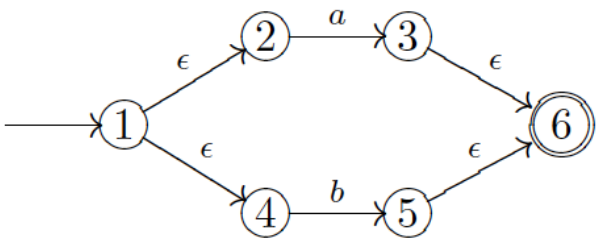
✓ 1010 x 2014 x 101 x 000 ✓ 0000

Langage accepté : soit A un automate fini
L(A) = {ω | ω est acceptée par A}

AFN

Pareil que un AFD sauf

- Σ = alphabet $\{\varepsilon, x_1, x_2, \ldots, x_n\}$
- $\delta \subseteq Q \times \Sigma \times Q \times \Sigma$



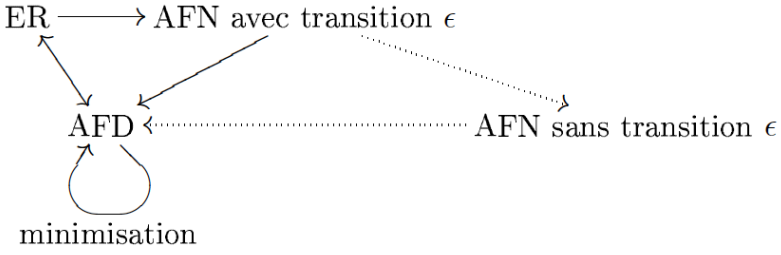
Expressions régulières

Une autre façon de définir un langage en intension.

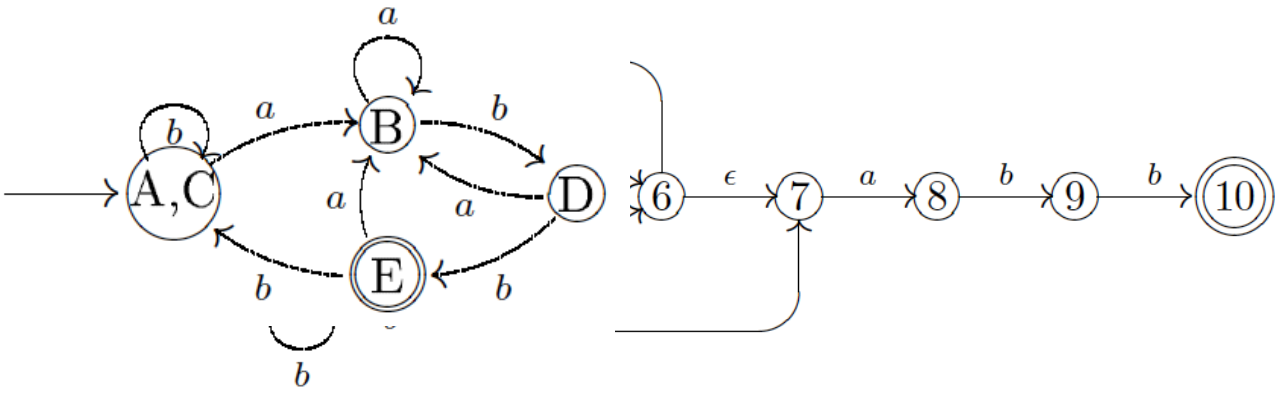
(a|b)*abb

✓ aaaabb ; babb ; abb…

x Bb ; aaaba ; bbbbb…



Conversions



Langages non-contextuels N → s, s ∈ (T∪N)*

L'automate à pile est aux grammaires non-contextuelles ce que les automates finis sont aux grammaires régulières.

Automate à pile (AP) : $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ tq.

- $Q = \{ q_0, \ldots, q_k \}$
- Σ = alphabet fini d'entrée
- Γ = alphabet fini de la pile
- $\delta : (Q \times \Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow Q \times \Gamma^*$
- $q_0 \in Q$ état initial
- Z_0 symbole initial dans la pile
- $F \subseteq Q$ états finaux

Automates à pile

Comportement d'acceptation :

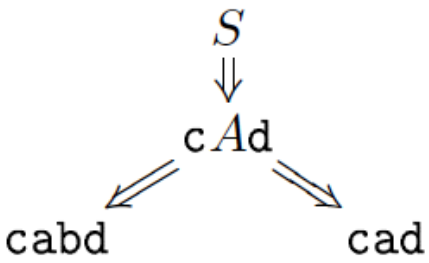
ω est accepté par l'automate AP si toute la chaîne ω est consommée, on arrive dans un état final et la pile est vide.

Comment construire l'automate à pile à partir d'une grammaire ?

- Méthode universelle (Cocke, Younger et Kasami) $O(n^3)$
- Analyse descendante LL(k)
- Analyse ascendante LR(k)

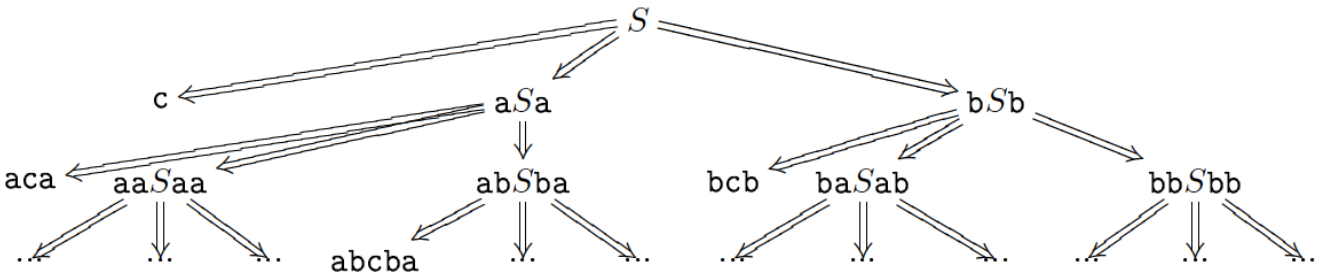
Analyse descendante

$$S \rightarrow \texttt{c}A\texttt{d}$$
$$A \rightarrow \texttt{ab} \mid \texttt{a}$$



Analyse descendante

$$S \rightarrow \mathtt{c} \mid \mathtt{a}S\mathtt{a} \mid \mathtt{b}S\mathtt{b}$$



① **Descente récursive non prédictive**
② **Descente récursive prédictive LL(k)**
k est le nombre de lexème à tester pour prendre une décision

LL(1)

- Un seul *token* suffit pour prendre la bonne décision
- Bon avec de grammaires petites
- Restriction :
  - -Récursion gauche interdite
  - -Nécessité de factorisation à gauche

Factorisation à gauche

S ⟶ if **E** then **S** | if **E** then **S** else **S**
E ⟶ …

Trouver le plus long préfixe α commun au deux alternatives
Si $\alpha \neq \varepsilon$ remplacer $A \longrightarrow \alpha\beta 1 \mid \alpha\beta 2 \mid \ldots \mid \alpha\beta n \mid \omega$
par :

$$A \longrightarrow \alpha A' \mid \omega$$
$$A' \longrightarrow \beta 1 \mid \beta 2 \mid \ldots \mid \beta n$$

S ⟶ if **E** then **S** **S′**
S′ ⟶ ε | else **S**
E ⟶ …

Elimination des récursivités à gauche

E ⟶ **E** + **E**

E ⟶ **E** + **E** + **E** + **E** + **E** …

S ⟶ **A**α
A ⟶ **S**

S ⟶ **A**α ⟶ **A**αα ⟶ **A**ααα ⟶ **A**αααα ⟶ **A**ααααα …

Elimination des récursivités directes

$$A \longrightarrow A\alpha 1 \mid \ldots \mid A\alpha m \mid \beta 1 \mid \ldots \mid \beta n$$

$$\begin{array}{lll} A & \longrightarrow & \beta 1 A' \mid \ldots \mid \beta n A' \\ A' & & \alpha 1 A' \mid \ldots \mid \alpha m A' \mid \varepsilon \end{array}$$

$$\begin{array}{lll} E & \longrightarrow & E + T \mid T \\ T & \longrightarrow & T * F \mid F \\ F & \longrightarrow & (E) \mid id \end{array}$$

$$\begin{array}{lll} E & \longrightarrow & TE' \\ E' & \longrightarrow & + TE' \mid \varepsilon \\ T & \longrightarrow & FT' \\ T' & \longrightarrow & * FT' \mid \varepsilon \\ F & \longrightarrow & (E) \mid id \end{array}$$

Elimination des récursivités

1. On ordonne les non-terminaux A_1, A_2, ..., A_n.
2. **pour** $i := 1$ jusqu'à n **faire**
3. **pour** $j := 1$ jusqu'à $i - 1$ **faire**
4. remplacer toutes les productions $A_i \rightarrow A_j\gamma$ par les productions $A_i \rightarrow \delta_1\gamma|\delta_2\gamma|...|\delta_k\gamma$, où $A_j \rightarrow \delta_1|\delta_2|...|\delta_k$ sont toutes les A_j-productions courantes.

fin

5. éliminer les récursivités à gauche immédiates des A_i-productions à l'aide de la transformation suivante :

$$A_i \rightarrow A_i\alpha_1|...|A_i\alpha_m|\beta_1|...|\beta_n$$

devient :

$$A_i \rightarrow \beta_1 A_i'|...|\beta_n A_i'$$
$$A_i' \rightarrow \alpha_1 A_i'|...|\alpha_m A_i'|\epsilon$$

fin

Elimination des récursivités

$$
\begin{aligned}
S &\rightarrow Aa \mid b \\
A &\rightarrow bdA' \\
A' &\rightarrow cA' \mid adA' \mid \varepsilon
\end{aligned}
$$

Analyse ascendante

LR(1) est l'approche utilisé presque partout

Plus souple car LL(1) $\subset$ LR(1)

EXPR	::=	*integer*	(R1)
	|	EXPR EXPR +	(R2)
	|	EXPR EXPR *	(R3)

((1 + 2) * 3) + 4 ≡ 1 2 + 3 * 4 +$

Analyse ascendante

Action	Input	Stack
	12+3*4+$	[]
Shift		
	2+3*4+$	[1]
Reduce (R1)		
	2+3*4+$	[Expr]
Shift		
	+3*4+$	[2Expr]
Reduce (R1)		
	+3*4+$	[Expr Expr]
Shift, Reduce (R2)		
	3*4+$	[Expr]
Shift, Reduce (R1)		
	*4+$	[Expr Expr]
Shift, Reduce (R3)		
	4+$	[Expr]
Shift, Reduce (R1)		
	+$	[Expr Expr]
Shift, Reduce (R2)		
	$	[Expr]

$$
\begin{array}{lcll}
\text{Expr} & ::= & \textit{integer} & \text{(R1)} \\
 & | & \text{Expr Expr +} & \text{(R2)} \\
 & | & \text{Expr Expr *} & \text{(R3)}
\end{array}
$$

Analyse ascendante (ambigüités)

$$
\begin{array}{lcll}
\text{E0} & ::= & \textit{integer} & \text{(R1)} \\
 & | & \text{E0 + E0} & \text{(R2)}
\end{array}
$$

Action	Input	Stack
⋮		
	+...	[E0 + E0 ...]
⋮		

Conflit **shift/reduce** : on empile le **+** ou on reduit **E0+E0** ?

Analyse ascendante (ambigüités)

$$
\begin{array}{lcll}
\mathrm{E1} & ::= & \mathit{integer} & (\mathrm{R1}) \\
 & | & \mathrm{E1} + \mathrm{E1} & (\mathrm{R2}) \\
 & | & \mathrm{E1} * \mathrm{E1} & (\mathrm{R3})
\end{array}
$$

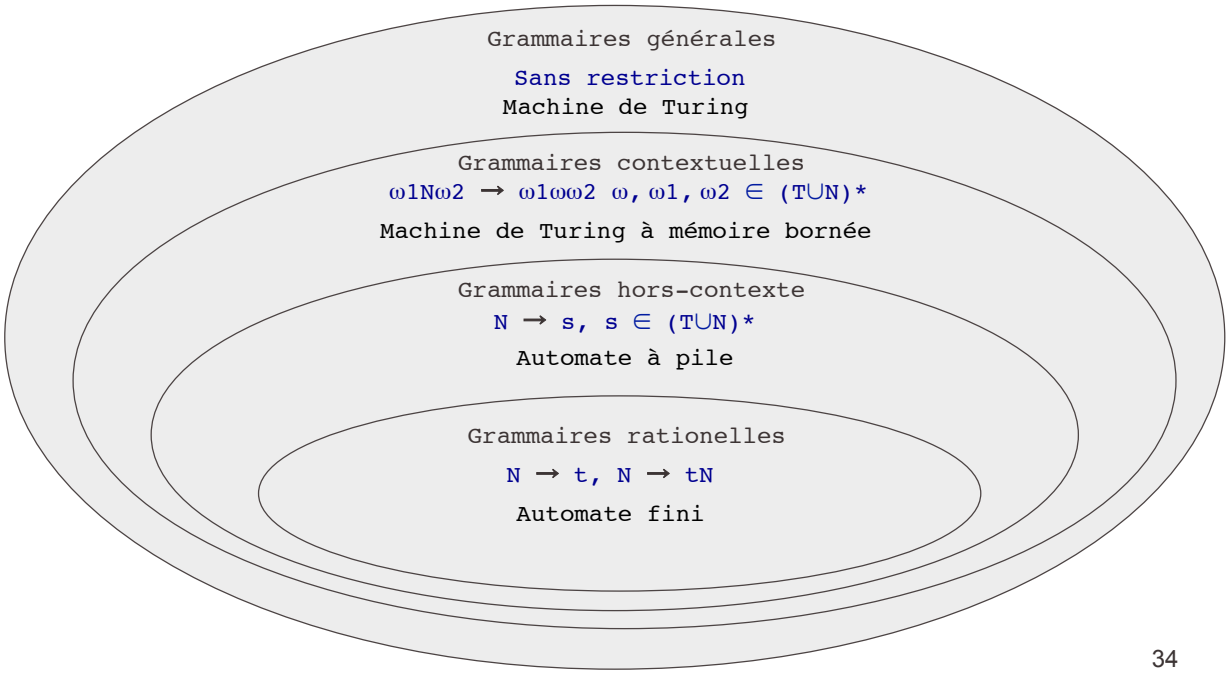
$$(\mathit{integer} + \mathit{integer}) * \mathit{integer} \neq \mathit{integer} + (\mathit{integer} * \mathit{integer})$$

On donne des priorités dans la grammaire…

… mais aussi peut transformer la grammaire.

$$
\begin{array}{lcll}
\mathrm{E} & ::= & \mathrm{E} + \mathrm{T} & (\mathrm{R1}) \\
 & | & \mathrm{T} & (\mathrm{R2}) \\
\mathrm{T} & ::= & \mathrm{T} * \mathrm{F} & (\mathrm{R3}) \\
 & | & \mathrm{F} & (\mathrm{R4}) \\
\mathrm{F} & ::= & \mathit{integer} & (\mathrm{R5})
\end{array}
$$

Hiérarchie de Chomsky



Pour les langages de programmation

Caractérisation des langages en fonction de la difficulté de reconnaissance d'une chaîne.

- **Forth, LISP**
- **ML, Pascal, C, etc**
- **...**
- **C++ Java**

Plus une grammaire est difficile, plus

- Augmente la complexité (temps / espace)
- Lisibilité des automates
- Possibilité d'ambigüité
- Messages d'erreur des parseurs lisibles.

- Lexique simple (textuel).
- Grammaire contextuelle dans la plus part des cas.

Backus-Naur Form (BNF)

<terme> ::= <nombre signé> | <nombre signé> <op> <terme>

<nombre signé> ::= <nombre> | « + » <nombre> | « - » <nombre>

<nombre> ::= <entier> | <nombre fractionnaire>

<nombre fractionnaire> ::= <entier> | <entier> « / » <entier>

<entier> ::= <chiffre> | <chiffre><entier>

<chiffre> ::= « 0 » | « 1 » | … | « 8 » | « 9»

<op> ::= « * » | « / » | … | « + » | « -»

méta-symboles + non terminaux + terminaux		
::= , I , (,)	<nom>	« mots »

Exemple d'un langage

Lexique {**1**, **0**}

Syntaxe

B ::= BC | C

C ::= **0** | **1**

Sémantique

$B_v : B \rightarrow Nat$

$B_v [[BC]] = (B_v [[B]] * 2) + C_v[[C]]$

$B_v [[C]] = C_v [[C]]$

$C_v : C \rightarrow Nat$

$C_v [[\mathbf{0}]] = 0$

$C_v [[\mathbf{1}]] = 1$

Exemple d'un langage

$B_v[[\mathbf{101}]] = (B_v [[\mathbf{10}]] * 2) + C_v [[\mathbf{1}]]$

$= ((B_v[[\mathbf{1}]] * 2))* 2 + C_v [[\mathbf{0}]]) + 1$

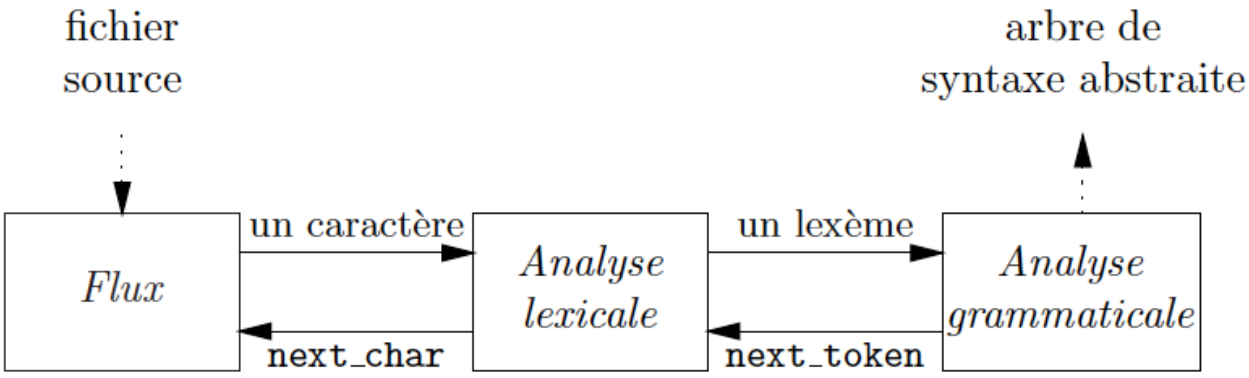
$= ((C_v[[\mathbf{1}]] * 2))* 2 + 0) + 1$

$= (1 * 2)* 2 + 0) + 1$

$= 5$

Outils d'analyse

Rapport entre les analyseurs



Analyse lexicale (Lex)

Transformer un suite de caractères en une suite de mots (lexèmes / *tokens*). Peut se faire au même temps que l'analyse grammaticale, mais…

Le but de l'analyseur lexical est donc de 'consommer' des symboles et de les fournir à l'analyseur syntaxique.

Un fichier de description pour Lex est formé de trois parties (optionnelles)

déclarations
%%
productions
%%
code additionnel

Lex (déclarations)

① Code dans le langage cible %{ %}
② Expressions régulières *non_terminal expression_regulière*

```
%{

#include "calc.h"

#include <stdio.h>
#include <stdlib.h>

%}


blancs    [\t\n ]+
lettre    [A-Za-z]
chiffre [0-9]
identificateur  {lettre}(_|{lettre}|{chiffre})*
entier  {chiffre}+
```

Lex (productions)

① Code dans le langage cible %{ %} (placé au début de *yylex()*)
② Productions *expression_régulière* *action*

```
%{
printf ("hello");
%}

[a-z]                printf ("%c", ((yytext[0]-'a'+13)%26)+'a');

[\r\n]|(\r\n)        {printf ("%s", yytext) ; fflush (stdout);
                      printf ("hola");
                      printf("%c", yytext [0]);}
```


Lex (code additionnel)

```
/* Code par default*/

main() {
  yylex();
}
```

Compil :

```
Flex myfile.lex && gcc lex.yy.c –lfl –o exe
```

Analyse grammaticale Yacc (Yet Another Compiler Compiler)

Produire le texte source d'un analyseur syntaxique du langage engendré par une grammaire du type LR(1)

Il est aussi possible, en plus de la vérification de la syntaxe de la grammaire, de lui faire effectuer des actions sémantiques.

Pareil que pour Lex, un fichier Yacc se compose de trois parties :

```
déclarations
%%
productions
%%
code additionnel
```


Yacc (déclarations)

① Code dans le langage cible %{ %}
② Déclaration des terminaux pouvant être rencontrés, grâce au mot-clé %token
③ Le type de donnée du terminal courant, avec le mot-clé %union.
④ Des informations donnant la priorité et l'associativité des opérateurs.
⑤ L'axiome de la grammaire, avec le mot-clé %start.

Yacc (productions)

① Déclarations et/ou définitions encadrées par %{ et %}
② Productions de la grammaire

```
notion_non_terminale:
corps_1 { action_semantique_1 }
| corps_2 { action_semantique_2 }
| ...
| corps_n { action_semantique_n }
;
```

Les corps_i peuvent être des notions terminales ou non terminales du langage.

Yacc (code additionnel)

Cette partie, qui comporte le code additionnel, devra obligatoirement comporter une déclaration du main() (qui devra appeler la fonction yyparse()), et de la fonction yyerror(char *message), appelée lorsqu'une erreur de syntaxe est trouvée.

Yacc (exemple)

```
%{
#include "global.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
%}

%token  NOMBRE
%token  PLUS  MOINS FOIS  DIVISE  PUISSANCE
%token  PARENTHESE_GAUCHE PARENTHESE_DROITE
%token  FIN

%left PLUS  MOINS
%left FOIS  DIVISE
%left NEG
%right  PUISSANCE

%start Input
%%
Input:
    /* Vide */
  | Input Ligne
  ;

Ligne:
    FIN
  | Expression FIN    { printf("Resultat : %f\n",$1); } ;

Expression:
    NOMBRE      { $$=$1; }
  | Expression PLUS Expression  { $$=$1+$3; }
  | Expression MOINS Expression { $$=$1-$3; }
  | Expression FOIS Expression  { $$=$1*$3; }
  | Expression DIVISE Expression  { $$=$1/$3; }
  | MOINS Expression %prec NEG  { $$=-$2; }
  | Expression PUISSANCE Expression { $$=pow($1,$3); }
  | PARENTHESE_GAUCHE Expression
PARENTHESE_DROITE  { $$=$2; }
  ;

%%

int yyerror(char *s) {
  printf("%s\n",s);
}

int main(void) {
  yyparse();
}
```


Yacc + Lex

```
%{
#include "global.h"
#include "calc.h"
#include <stdlib.h>
%}

blancs    [ \t]+
chiffre   [0-9]
entier    {chiffre}+
exposant  [eE][+-]?{entier}
reel    {entier}("."{entier})?{exposant}?

%%
{blancs}  { /* On ignore */ }
{reel}    {
      yylval=atof(yytext);
      return(NOMBRE);
    }

"+"   return(PLUS);
"-"   return(MOINS);
"*"   return(FOIS);
"/"   return(DIVISE);

"^"   return(PUISSANCE);

"("   return(PARENTHESE_GAUCHE);
")"   return(PARENTHESE_DROITE);

"\n"  return(FIN);
```

Compilation (Demo)

```
>bison -d calc.y
>mv calc.tab.h calc.h
>mv calc.tab.c calc.y.c
>flex calc.lex
>mv lex.yy.c calc.lex.c
>gcc -c calc.lex.c -o calc.lex.o
>gcc -c calc.y.c -o calc.y.o
>gcc -o calc calc.lex.o calc.y.o -ll
```

L'appel à bison (le Yacc de GNU) calc.tab.h pour définir les terminaux. On appelle flex (le Lex de GNU) et on donne des noms un peu plus convenables pour le tout. Il suffit alors de compiler, sans oublier la librairie spéciale Lex ("-ll").

```
>calc
 1+2*3
 Resultat : 7.000000
 2.5*(3.2-4.1^2)
 Resultat : -34.025000
```


ocamllex et ocamlyacc

Ocamllex format du fichier

```
{
Code Ocaml exécuté avant
}

Let <nom> = <expreg>

Rule <nom> = parse
<expreg>
| <expreg>          { <code action> }
…
| <expreg>          { <code action> }

{
Code Ocaml exécuté après
}
```


Ocamllex exemple

```
{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 -> ()
  }
```

Compil : `ocamllex file.mll && Ocamlopt file.ml –o exe`

Ocamllex avec un parseur

```
{
   open Parser
(* Parser doit définir type token = INT of int | OP of string | END *)
}


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


Ocamllex Demo

Références

- « Automates et langages » Support de cours de J. Malenfant
- Slides Compilation Avancée (MI190) Benjamin Canu
- « Compilers Principles, Techniques and Tools » Aho, Sethi et Ulman
- « Développement d'applications avec Ocaml » E.CHAILLOUX, P. MANOURY, B.PAGANO
- « Tutorial de Lex/Yacc » Etienne Bernard