AGP: Algorithmique et programmation

tanguy.risset@insa-lyon.fr
Lab CITI, INSA de Lyon
Version du July 22, 2016

Tanguy Risset

July 22, 2016

Langages, Grammaires, et Compilateurs

- Objectif de cette partie du cours: comprendre le mécanisme de compilation, le lien avec la notion de grammaire et savoir créer un parseur.
- A quoi est du le succès des ordinateurs?
  - Progrès technologique pour l'intégration de transistors
  - Augmentation de la productivité des programmeurs
    - Langage de haut niveau
    - Compilateurs rapides, codes portables
- La notion de compilation n'est pas limitée aux langages de programmation.
- L'informaticien produit en permanence des programmes qui font passer les données d'une représentation à une autre.

Plan

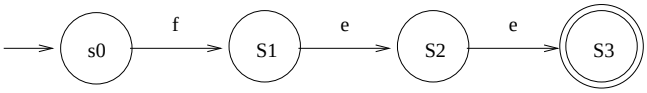
Sources:

- Cours Compilation T. Risset,
- "Engineering a compiler, Cooper & Torczon"
- `http://ds9a.nl/lex-yacc/`

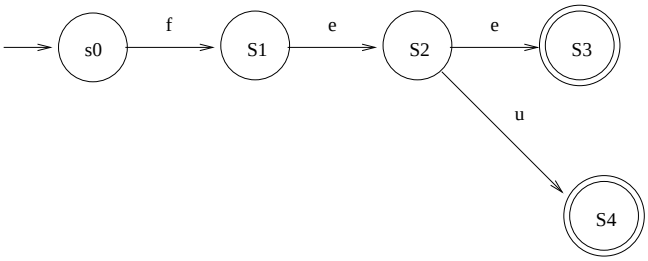
Table of Contents

Langage régulier et automate

- Formellement, un *langage* (au sens *syntaxe*) est simplement un ensemble de mots formés à partir d'un alphabet fini.
- Exemples de langage
  - l'ensemble des mots du dictionnaire (alphabet de `a` à `z`)
  - l'ensemble des mots constitués de deux `'a'` suivis d'un nombre quelconque de `'b'`: $\mathcal{L}=\{aa, aab, aabb, aabbb, \ldots\}$
- Un langage est dit *régulier* s'il est reconnu par un automate.
- Automate reconnaissant le mot `fee`:

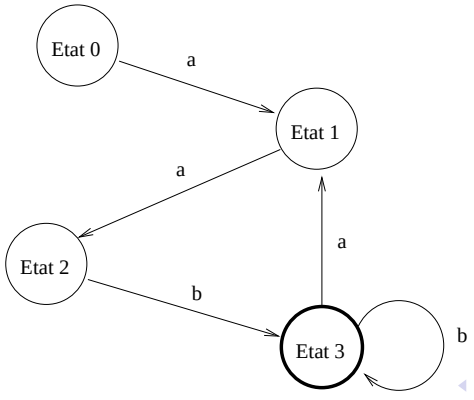


- Automate reconnaissant le langage {`fee`, `feu`}

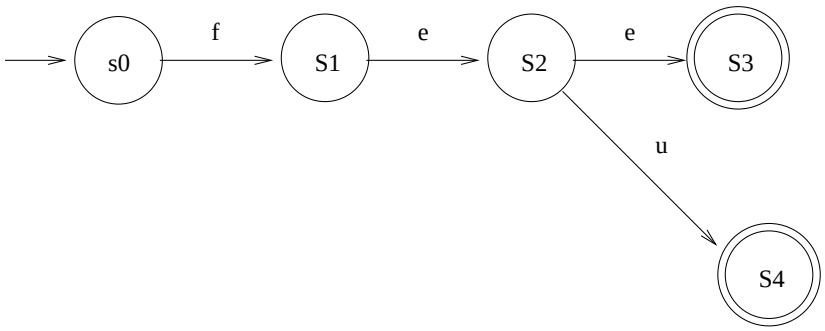


Notion d'automate

- Un automate est une collection de K états numérotés de 0 à K-1, ainsi qu'une collection de transitions
- Un état particulier est l'état initial.
- Tous les états sont soit des états d'acceptation et soit des états de refus
- Les transitions, sont des triplets (état 1, lettre `x`, état 2) qui signifient : lorsque je suis dans l'état 1 et que je lis la lettre `x`, alors je vais dans l'état 2.



Notion de mot reconnu

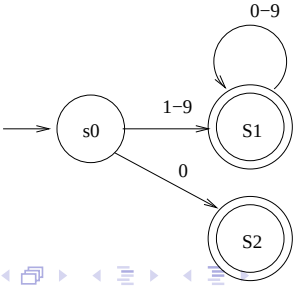


- `fee` → reconnu
- `feu` → reconnu
- `fei` → non reconnu (impossible de lire `'i'`)
- `fe` → non reconnu (arrêt dans un état non final)

Automate: définition formelle

- Un automate fini déterministe est donné par $(S, \Sigma, \delta, s_0, S_F)$ ou:
  - S est un ensemble fini d'états;
  - Σ est un alphabet fini;
  - $\delta : S \times \Sigma \rightarrow S$ est la fonction de transition;
  - s_0 est l'état initial;
  - S_F est l'ensemble des états finaux
- pour l'automate reconnaissant "fee", on a $S = \{s_0, s_1, s_2, s_3\}$, $\Sigma = \{f, e\}$ (ou tout l'alphabet)
  $\delta = \{\delta(s_0, f) \rightarrow s_1, \delta(s_1, e) \rightarrow s_2, \delta(s_2, e) \rightarrow s_3\}$.
  - Il y a en fait un état implicite d'erreur (s_e) vers lequel vont toutes les transitions qui ne sont pas définies.
  - Un automate *accepte* une chaîne de caractère si et seulement si en démarrant de s_0, il s'arrête dans un état final

automate qui reconnaît n'importe quel entier:



Expressions régulières

- Les expressions régulières sont couramment employées en ligne de commande, par exemple: `ls *.c`
- Une *expression régulière* `X` basée sur un alphabet décrit un *langage*, c'est à dire un ensemble de mots sur cet alphabet, on note ce langage `E(X)`.
- Exemples:

Expression régulière	langage reconnu
a*.b	mots constitués d'un nombre quelconque de a suivi d'un b
a.(a+b)*.a + b.(a+b)*.b	mots constitués des lettre a et b, commençant et finissant par la même lettre
a.(a+b)*.a+a	mots commençant par a et finissant par a (alphabet {a,b})

Expressions régulières: Définition formelle

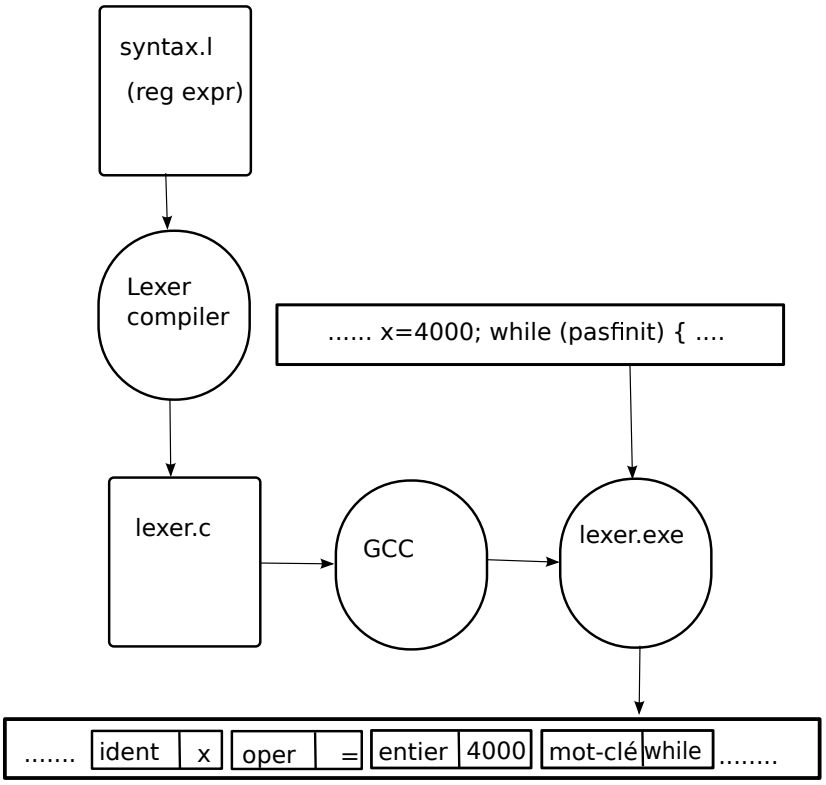
- Individuellement, chaque lettre est une expression régulière: `E(a)={a}`
- Si `X1` et `X2` sont deux expressions régulières, alors `X1.X2` est une expression régulière (concaténations des mots des deux langages).
- Si `X1` et `X2` sont deux expressions régulières, alors `X1+X2` est une expression régulière (union des mots des deux langages: `E(X1+X2)=E(X1)`$\cup$`E(X2)`)
- Si `X` est une expression régulière alors `X*` est une expression régulière qui décrit l'ensemble des mots construits en répétant autant de fois que l'on veut (éventuellement zéro fois) un mot décrit par `X`
- * plus prioritaire que . qui est plus prioritaire que +

Expression régulière et analyse lexicale

- Les expressions régulières définissent une classe de langage simple (langages réguliers) reconnue par des automates finis.
- On va utiliser cela en compilation pour identifier les *tokens* (ou *lexèmes*) du langage, c'est à dire les unités lexicales élémentaires:
  - les mots-clés (`do`, `while`, `for`, etc.)
  - les identificateurs (noms de variable, de fonction, etc.)
  - les constantes (entier, flottant, chaîne de caractère)
- C'est la première phase de la compilation: ***l'analyse lexicale***
- Il existe depuis longtemps des outils (historiquement: `lex`) qui, à partir des expressions régulières décrivant les lexèmes, génèrent automatiquement le programme qui reconnaît les lexèmes.
- C'est la notion de compilateur de compilateur.

Table of Contents

Principe d'analyse lexicalle (*lexer*)



Exemple d'analyseur lexical: `lex`

- Lex (ou flex sa version actuelle pour linux), produit un analyseur lexical, c'est à dire un programme qui prend en entrée un flot de caractères et produit certaines actions en fonction.
- Par défaut, un analyseur lexical recopie son entrée standard sur la sortie standard, et en plus il fait les actions spécifiées dans le fichier de configuration.
- Voici un exemple simple de fichier de configuration: l'analyseur lexical décrit affiche un message quand il voit les mots "`start`" ou "`stop`" passer dans le flux de caractères. (fichier `exemple1.l`):

```
%{
#include <stdio.h>
%}

%%
stop    printf("Stop command received\n");
start   printf("Start command received\n");
%%
```

Utilisation de lex

- Soit le fichier `exemple1.l`:

```
%{
#include <stdio.h>
%}

%%
stop    printf("Stop command received\n");
start   printf("Start command received\n");
%%
```

- on le compile avec la commande:
  `flex example1.l`

→ fichier `lex.yy.c` généré...
n'essayez pas de le comprendre!!

- Compilation du lexer:
  `gcc lex.yy.c -o example1 -ll`

```
trisset@fania:~$ ./example1
jdsdf
jdsdf
start
Start command received

startuuuuu
Start command received
uuuuu
stoprtr
Stop command received
rtr
\^D
trisset@fania:~$ ./example1
```


Autre exemple lex: tokenisation

- Soit le fichier `exemple2.l`:

```
%{
#include <stdio.h>
%}

%%
[0123456789]+ printf("NUMBER\n");
[a-zA-Z][a-zA-Z0-9]* printf("WORD\n");
%%
```

- `flex example2.l`
  `gcc lex.yy.c -o example2 -ll`
- Le flot de caractères est transformé en un flot de *tokens* (composant syntaxique, ou lexèmes), on appelle cela la *tokenisation*
- Le flot de tokens peut à son tour être analysé par un *analyseur syntaxique*

```
trisset@fania:~$ ./example2
foo
WORD

bar
WORD

123
NUMBER

bar123
WORD

123bar
NUMBER
WORD
```

A retenir concernant `lex`

- `lex` est un outil transformant un flot de **caractères** en un flot de **tokens**
- Les tokens sont reconnus grâce à des **expressions régulières**
- La syntaxe des expressions régulières de lex est un peu spéciale...
- `lex` prend en entrée un fichier de configuration **dont l'extension est .l** et produit un fichier C (par défault **lex.yy.c**)
- Le fichier C produit par lex contient une fonction particulière **yylex()**, c'est elle qui effectue l'analyse lexicale des caractères arrivant sur l'entrée standard.
- Par défaut la fonction `yylex()` retourne (i.e. l'analyseur lexical s'arrête) lorsqu'il rencontre le caractère `fin-de-fichier` (Ctrl-D ou ^D sur l'entrée standard).
- On peut aussi arrêter l'analyse lexicalle en reconnaissant un token, en mettant l'instruction `return 0;`

Renvoyer les tokens

- `yylex()` peut aussi **renvoyer des tokens**
- cette fonction va être appelée repétitivement par `yacc` pour obtenir tout les tokens un à un
- `yacc` *définit* des token qui sont ensuite *reconnus* par `lex`
- Les deux variables `yytext` et `yyval` sont utilisées pour communiquer entre `lex` et `yacc`:
  - `yytext` est une chaine de caractères qui contient les caractèresstoken du token reconnu.
  - Si ce token a une valeur (par exemple un entier), lex met cette valeur dans `yyval`

Renvoyer les tokens: exemple

- Ci dessous un programme `lex` qui identifie les tokens `NUMBER`, `PLUS` et `LETTRE` qui sont définis par ailleurs (par `yacc`)
- `yytext` est mise à jour par défaut.
- on utilise du C (ici `atoi`) pour mettre `yyval` à sa valeur

```
%{
#include <stdio.h>
#include "adder.tab.h"
%}
%%
[a-zA-Z]                return LETTRE;
[0-9]+                    yylval=atoi(yytext); return NUMBER;
\+                        return PLUS;
\n                        /* ignore linebreak */;
[ \t]+                    /* ignore whitespace */;
%%
```


Table of Contents

Définition d'un langage de programmation

- Langage programmation = Syntaxe + Sémantique
- La syntaxe d'un langage définit l'ensemble des règles qui décrivent la structure de ce langage.
- Elle est définie par une *grammaire* qui elle même se décrit par un langage appelé *métalangage*.
- Un métalangage est un langage qui permet de décrire un autre langage, par exemple la notation Backus-Naur Form (BNF)

Notation BNF

Exemple de méta-langage: La notation Backus-Naur Form (BNF)

- Les symboles <, >, et ::= sont ceux du métalangage et n'appartiennent pas au langage décrit.
- Le symbole ::= signifie «est défini par».
- Les symboles placés entre les chevrons < et > sont des symboles non-terminaux (ils appartiennent au métalangage)
- Les autres symboles sont appelés les symboles terminaux.

```
<terme> ::= <terme> * <chiff
<terme> ::= <chiffre>
<chiffre> ::= 0
<chiffre> ::= 1
<chiffre> ::= 2
...
<chiffre> ::= 9
```

Grammaire et langage

- Une grammaire permet de définir un langage:

  `<terme> ::= <terme> * <chiffre> | <chiffre>`
  `<chiffre> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`
- En partant du non-terminal `terme`, on peut décrire les multiplications d'un nombre quelconque de chiffres, par exemple `0*9*7` ou `8*8*8*8`. Ce sont des *mots* du *langage* définit par la grammaire.
- Les règles ci dessus sont appelées *règles de productions* de la grammaire.
- Pour définir complètement une grammaire, il faut un quadruplet: `(S,T,N,P)`
  - `S` est le symbole de départ,
  - `T` est l'ensemble des terminaux
  - `N` est l'ensemble des non-terminaux
  - `P` est l'ensemble des règles de production

Exemple: expression arithmétique

- le langage considéré est celui des expressions arithmétiques (un mot de ce langage: $2 + 3 * 6 - 7/8$)
- Considérons la grammaire suivante:

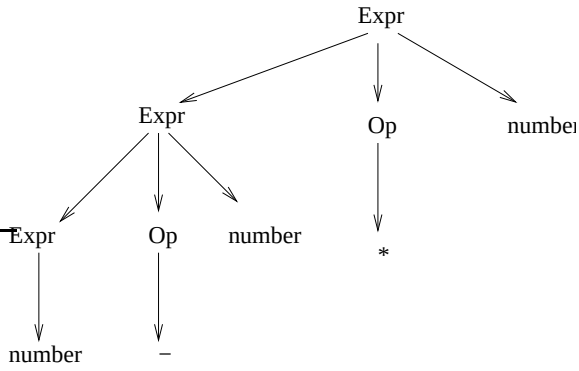
1.	$< Expr >$	$\rightarrow$	$< Expr > \ < Op > \ NUMBER$
2.		$\rightarrow$	$NUMBER$
3.	$< Op >$	$\rightarrow$	$+$
4.		$\rightarrow$	$-$
5.		$\rightarrow$	$\times$
6.		$\rightarrow$	$\div$

- les symboles qui ne sont pas entre chevrons ($<$ et $>$) sont les terminaux du langage.
- Notons ici que les *terminaux* du langages correspondent aux *tokens* que nous avons identifiés par l'analyse lexicale.

Arbre de dérivation

- On cherche à reconnaître l'expression $12 - 45 \times 19$
- En appliquant la séquence de règles: 1, 5, 1, 4, 2 on arrive à

Regle	*Phrase*
	$< Expr >$
1	$< Expr > < Op >$ *number*
5	$< Expr >$ $\times$ *number*
1	$< Expr > < Op >$ *number* $\times$ *number*
4	$< Expr >$ $-$ *number* $\times$ *number*
2	*number* $-$ *number* $\times$ *number*



- On represente cette dérivation par un *arbre de dérivation* (ou arbre syntaxique, arbre de syntaxe, parse tree).

Autre dérivation?

- On a toujours utilisé la règle du non-terminal le plus à droite (*rightmost derivation*)
- En appliquant les règles 1, 1, 2 ,4 , 5, on obtient:

Règle	*Phrase*
	Expr
1	$< Expr > < Op >$ *number*
1	$< Expr > < Op >$ *number* $< Op >$ *number*
2	*number* $< Op >$ *number* $< Op >$ *number*
4	*number* $-$ *number* $< Op >$ *number*
5	*number* $-$ *number* $\times$ *number*

- L'arbre de dérivation reste le même, on dit que la grammaire est ***non-ambiguë***.

Autre grammaire pour le même langage

- Soit la grammaire suivante:

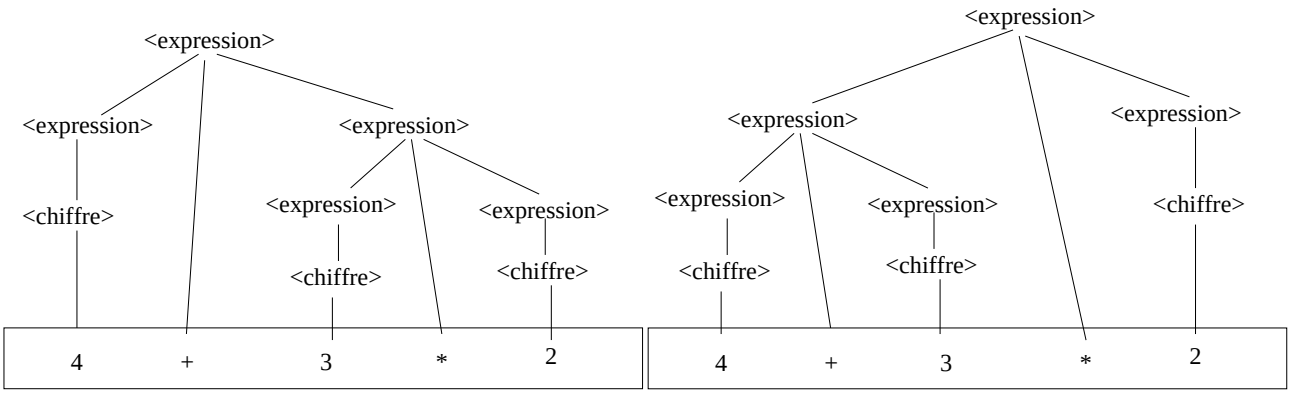
```
<expression> ::= <expression> + <expression> |
                 <expression> * <expression> |
                 <number>
<number> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

- Le mot 4+3*2 fait partie du langage de cette grammaire, il correspond à la dérivation:

```
<expression> ==>  <expression> + <expression>
     ==>  <expression> + <expression> * <expression>
     ==>   <number> + <expression> * <expression>
     ==>   <number> + <number> * <expression>
     ==>   <number> + <number> * <number>
     ==>   4 + <number> * <number>
     ==>   4 +  3 * <number>
     ==>   4 + 3 * 2
```


Arbre de dérivation pour 4+3*2

- Deux arbres possibles: la grammaire est ambiguë.



- Deux dérivations sont équivalentes si elles ont le même arbre de dérivation (seul l'ordre dans lequel on a choisit les règles peut changer).
- Pour nous, l'expression n'est pas ambiguë, il s'agit bien de 4+(3*2)
- Les grammaires manipulées ne doivent pas être ambiguës.

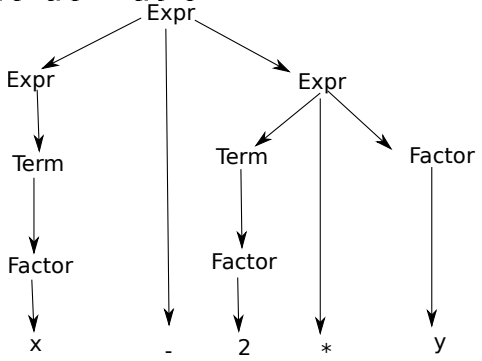
Une bonne grammaire pour les expressions arithmétiques

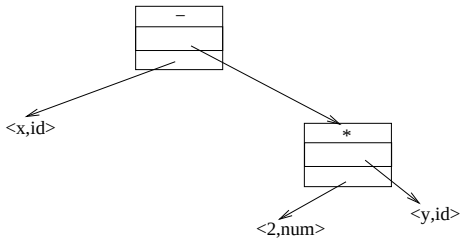
- Non-ambiguë,
- Respecte les priorité des opérateurs (L'arbre de dérivation de $12 - 45 \times 19$ représente bien $12 - (45 \times 19)$ et non pas $(12 - 45) \times 19$.),
- Respecte l'associativité.

1.	*Expr*	$\rightarrow$	*Expr* $+$ *Term*
2.		$\rightarrow$	*Expr* $-$ *Term*
3.		$\rightarrow$	*Term*
4.	*Term*	$\rightarrow$	*Term* $\times$ *Factor*
5.		$\rightarrow$	*Term* $\div$ *Factor*
6.		$\rightarrow$	*Factor*
7.	*Factor*	$\rightarrow$	(*Expr*)
8.		$\rightarrow$	*number*
9.		$\rightarrow$	*identifier*

Lien entre dérivation et représentation interne

- Pour être manipulé dans un programme, l'expression $x - 2 \times y$ doit être stockée dans une structure arborescente qui ressemble à l'arbre de dérivation:





- Un ***parseur*** ou analyseur syntaxique est un programme qui permet de reconstruire, l'arbre de dérivation d'un mot à partir de sa grammaire.
- Le parseur va aussi être utilisé pour **construire la représentation interne du programme**.

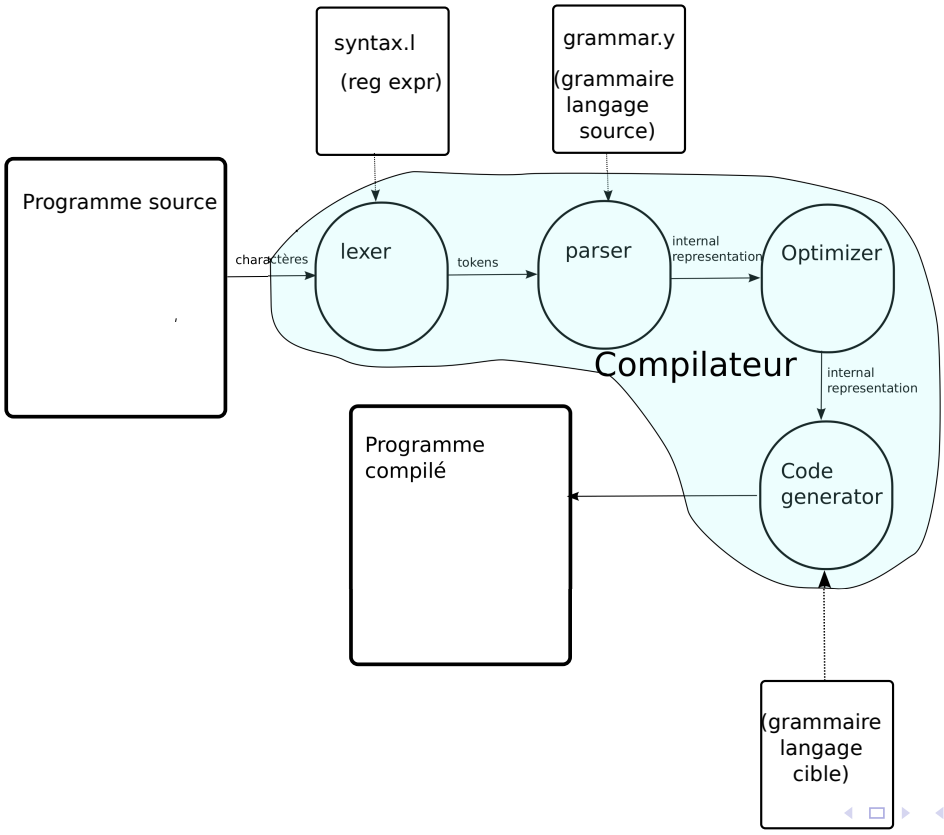
Prenons un peu de recul

- On a:
  - les analyseurs lexicaux (*lexer*), qui construisent une suite de tokens à partir d'une suite de caractères.
  - les analyseurs syntaxtiques (*parser*) qui, à partir de la grammaire et d'une suite de tokens permettent de reconstruire l'arbre de dérivation.
- En combinant les deux: d'abord le lexeur puis le parseur, on construit un compilateur, qui à partir d'une suite de caractères reconstruit l'arbre de dérivation qui va nous servir de représentation intermédiaire de notre programme pour générer un autre format (binaire ou autre format source)

Table of Contents

l'intérieur d'un Compilateur

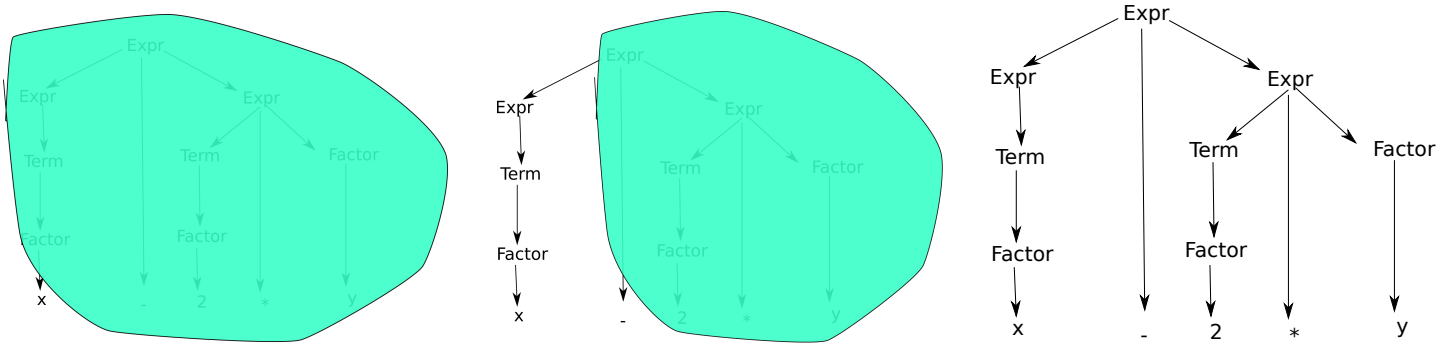
- Lexeur et Parseur sont les deux premières *passes* de compilation



Analyse syntaxique (*parsing*)

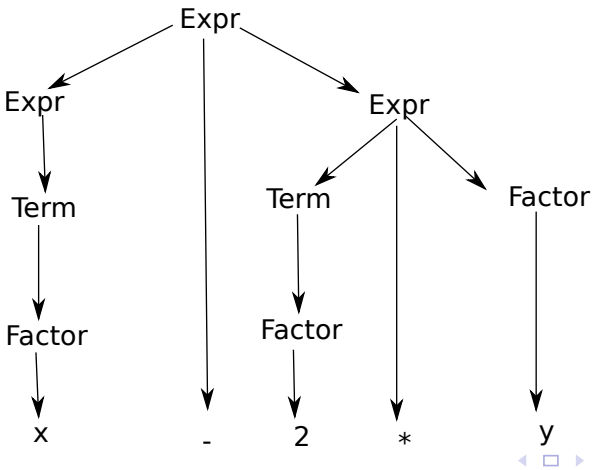
- On s'intéresse aux grammaires *non contextuelles* (celles qu'on a vu jusqu'à présent), plus puissantes que les grammaires régulières, mais pas assez pour analyser la langue naturelle.
- Un analyseur syntaxique doit retracer le cheminement d'application des règles de syntaxe qui ont menés de l'axiome au texte analysé.
- Il existe deux méthodes pour cela:
  - Une analyse *descendante* retrace cette dérivation en partant de l'axiome et en essayant d'appliquer les règles pour retrouver le texte.
  - Une analyse *ascendante* retrouve ce cheminement en partant du texte, en tentant de *réduire* les règles jusqu'à retrouver l'axiome.
- Le succès des compilateurs est lié au fait qu'on peut reconstruire cet arbre syntaxique en regardant peu de tokens (un seul idéalement) sur l'entrée standard (voir notamment au travaux de Donald Knuth sur l'analyse LR(k))

analyse syntaxique ascendante (LR(1))



Exemple d'analyse syntaxique ascendante (LR(1))

- On cherche à reconnaître $x - 2 \times y$ avec la grammaire des expressions arithmétique.
- $\uparrow$ désigne l'endroit ou se situe le curseur de lecture.
- A chaque étape on indique si on avance le curseur (shift) ou si on utilise une réduction (reduce) par une règle de la grammaire
- On veut reconstruire cet arbre:



Exemple d'analyse syntaxique ascendante (LR(1))

Action	*Règle utilisée*	*Etat*
		$\uparrow x - 2 \times y$
shift	*Identifier*	*Identifier* $\uparrow -2 \times y$
reduce	*Factor* $\rightarrow$ *Identifier*	*Factor* $\uparrow -2 \times y$
reduce	*Term* $\rightarrow$ *Factor*	*Term* $\uparrow -2 \times y$
reduce	*Expr* $\rightarrow$ *Term*	*Expr* $\uparrow -2 \times y$
shift	$-$	*Expr* $-\uparrow 2 \times y$
shift	*Number*	*Expr* $-$ *Number* $\uparrow \times y$
reduce	*Factor* $\rightarrow$ *Number*	*Expr* $-$ *Factor* $\uparrow \times y$
reduce	*Term* $\rightarrow$ *Factor*	*Expr* $-$ *Term* $\uparrow \times y$
shift	$\times$	*Expr* $-$ *Term*$\times \uparrow y$
shift	*Identifier*	*Expr* $-$ *Term* $\times$ *Identifier* $\uparrow$ (*EOF*)
reduce	*Factor* $\rightarrow$ *Identifier*	*Expr* $-$ *Term* $\times$ *Factor* $\uparrow$ (*EOF*)
reduce	*Term* $\rightarrow$ *Term* $\times$ *Factor*	*Expr* $-$ *Term* $\uparrow$ (*EOF*)
reduce	*Expr* $\rightarrow$ *Expr* $+$ *Term*	*Expr* $\uparrow$ (*EOF*)
success		

Comment ça marche

- Nous ne détaillons pas ici la théorie de l'analyse syntaxique.
- Cette théorie prouve que l'on peut choisir la bonne règle à chaque fois, en regardant simplement le prochain caractère à lire.
- Ce qu'il faut savoir pour utiliser yacc:
  - La technique utilisée est dite *LR(1)* (ou *shift/reduce parsing*), Ces parseurs lisent de gauche à droite et construisent (à l'envers) une dérivation la plus à droite en regardant au plus un symbole sur l'entrée, d'où leur nom: *Left-to -right scan, Reverse-rightmost derivation with 1 symbol lookahead*.
  - La grammaire ne doit pas être ambiguë
  - Lors de chaque réduction, yacc réalise des *actions* qui permettent de construire, *en parsant*, la représentation intermédiaire du programme.

Exemple d'action possible

S/R	*Règle utilisée*	*Action*	*Etatdel'analyse*
			$\uparrow x - 2 \times y$
shift	*Identifier*	Creer nœud Id	*Identifier* $\uparrow -2 \times$
reduce	*Factor* $\rightarrow$ *Identifier*		*Factor* $\uparrow -2 \times y$
reduce	*Term* $\rightarrow$ *Factor*		*Term* $\uparrow -2 \times y$
reduce	*Expr* $\rightarrow$ *Term*		*Expr* $\uparrow -2 \times y$
shift	$-$		*Expr* $-\uparrow 2 \times y$
shift	*Number*	Creer nœud Num	*Expr* $-$ *Number*
reduce	*Factor* $\rightarrow$ *Number*		*Expr* $-$ *Factor* $\uparrow$
reduce	*Term* $\rightarrow$ *Factor*		*Expr* $-$ *Term* $\uparrow$
shift	$\times$		*Expr* $-$ *Term*$\times \uparrow$
shift	*Identifier*	Creer nœud Id	*Expr* $-$ *Term* $\times$
reduce	*Factor* $\rightarrow$ *Identifier*		*Expr* $-$ *Term* $\times$
reduce	*Term* $\rightarrow$ *Term* $\times$ *Factor*	Creer nœud $\times$	*Expr* $-$ *Term* $\uparrow$ (
reduce	*Expr* $\rightarrow$ *Expr* $+$ *Term*	Creer nœud -	*Expr* $\uparrow$ (*EOF*)
success			

Résultat du l'analyse syntaxique

L'analyse syntaxique construit un AST (Abstract Syntax Tree), en même temps qu'elle trouve l'arbre de dérivation du mot.

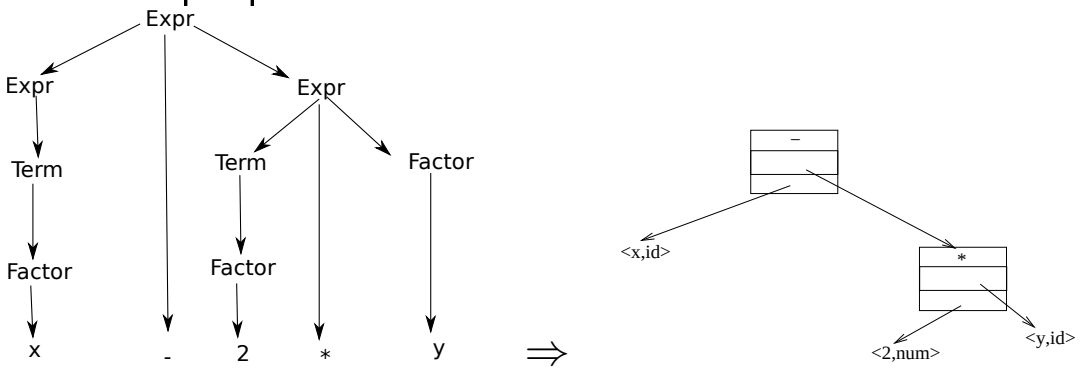


Table of Contents

Yacc (bison)

- `flex` est l'implémention de `lex` par Vern Paxson et `bison` est la version `GNU` de `yacc`, dans la suite on parlera toujours de `lex` et `yacc` pour désigner `flex` et `bison`
- `yacc` signifie *Yet Another Compiler Compiler*
- `yacc` peut parser des flots de *tokens*, il doit donc être utilisé avec un front-end qui transforme un flot de caractères en un flot de tokens (par exemple `lex`)

Premier exemple simple: un petit additionneur

- On doit lire une suite d'addition et afficher le résultat.
  - entrée du parser: 3+4+6
  - affichage du parseur: `résultat: 13`
- Proposition de grammaire élémentaire:
  - Grammaire (S,T,N,P),
  - S={ SOM },
  - N={ EXPR, SOM }
  - T={ ENTIER, '+' }
  - P={
    ```
    SOM ::= EXPR
    EXPR ::= NUMBER |
             EXPR '+' NUMBER
    ```
    }

Additionneur simple: ficher adder.l

```
%{
#include <stdio.h>
#include "adder.tab.h"
%}
%%
[0-9]+                  yylval=atoi(yytext); return NUMBER;
\+                      return PLUS;
\n                      /* ignore linebreak */;
[ \t]+                  /* ignore whitespace */;
%%
```

- `adder.tab.h` sera généré par la commande `yacc`, à inclure dans le fichier `lex.yy.c`
- `yyval` et `yytext` sont des variables partagées par `lex` et `yacc`: quand `lex` reconnaît un lexème, il le met dans la variable `yytext` et le transmet à `yacc`. Quand c'est une valeur, `yacc` s'attend à le trouver dans `yylval`

Additionneur simple: grammaire ⇒ adder.y

```
Rappel de  la grammaire:

SOM ::= EXPR
EXPR ::=  NUMBER |
          NUMBER '+' EXPR
```

```
//fichier adder.y
%%
SOM: EXPR
{
  fprintf(stdout,"somme: %d\n",$1);
  $$=$1;
}

EXPR:   NUMBER
        {
          $$=$1;
        }
      | NUMBER PLUS EXPR
        {
          $$=$1 + $3;
        }

%%
main()
  {
      yyparse();
  }
```

- A chaque réduction de règles est associée une action
- chaque symbole renvoie un objet
- $1, $2,... correspondent aux objets renvoyés par la partie droite de la règles. $$ correspond à l'objet renvoyé par la partie gauche

Additionneur simple: ficher adder.y complet

```
%{
#include <stdio.h>
#include <string.h>

void yyerror(const char *str)
{
    fprintf(stderr,"error: %s\n",str);
}

int yywrap()
{
     return 1;
}

%}


%token NUMBER
%token PLUS

//suite à droite
//....
```

```
//suite de la gauche
%%
SOM: EXPR
     {
       fprintf(stdout,"somme: %d\n",$1);
       $$=$1;
      }

EXPR:   NUMBER
        {
          $$=$1;
        }
      | NUMBER PLUS EXPR
        {
          $$=$1 + $3;
        }

%%
main()
  {
      yyparse();
  }
```

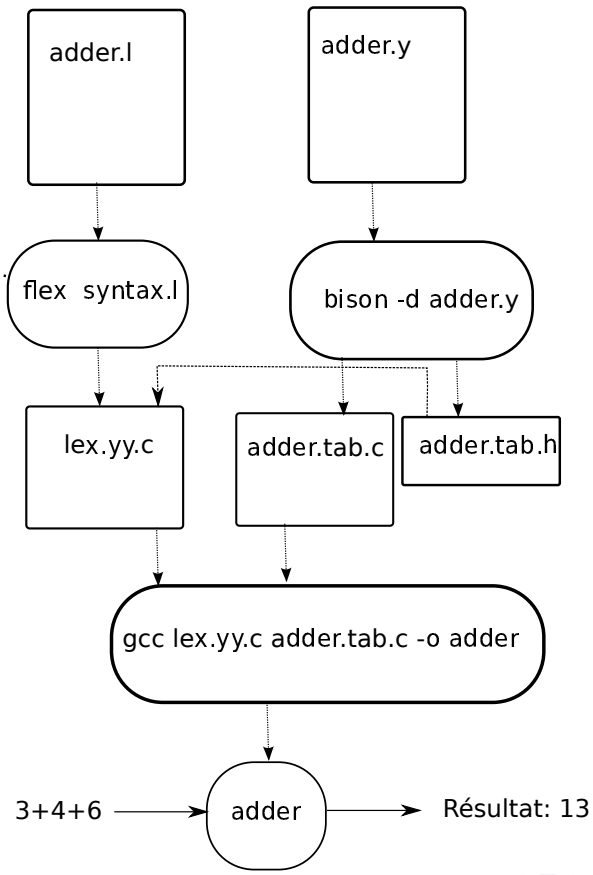
- A chaque réduction de règles est associé une action
- chaque symbole renvoie un objets
- $1, $2,... correspondent aux objets renvoyés par la partie droite de la règles. $$ correspond à l'objet renvoyé par la partie gauche

Structure d'un fichier yacc/bison

```
... definition ...
%%
....règles...
%%
...code...
```

- Définitions:
  - Code C copié au début du fichier `.c` généré (entre %{ et %})
  - On doit notamment définir `yyerror` et `yywrap`
  - définition des tokens (type, associativité éventuellement)
- Règles:
  - définition des règles de grammaire avec les actions associées aux réductions
- Code:
  - Le code que vous rajoutez, au minimum un `main` qui appelle `yyparse` comme ici

Adder: Processus de compilation global



Compilation et utilisation du programme adder

compilation:

```
trisset@fania:~/$ make
bison -d adder.y
gcc    -c -o adder.tab.o adder.tab.c
flex adder.l
gcc    -c -o lex.yy.o lex.yy.c
gcc adder.tab.o lex.yy.o -o adder
trisset@fania:~/$
```

exécution:

```
trisset@fania:~/$ ./adder
2+3
^D
somme: 5
trisset@fania:~/$ ./adder
1+2
+7

+9
^D
somme: 19
```


Le Langage TC-LOGO

- Le langage TC-LOGO (version 1.0) est hérité du formalisme LOGO (`http://en.wikipedia.org/wiki/Logo_programming_language`) destiné à dessiner un graphique.
- Il sert à décrire le chemin que va suivre le crayon pour dessiner ce graphique.
- Il comporte 4 instructions:
  - `FORWARD` *n*
  - `LEFT` *n*
  - `RIGHT` *n*
  - `REPEAT` [*programme TC-LOGO*]
- Il n'y a pas de point-virgule en TC-LOGO, les séparateurs valides sont l'espace, le retour chariot ou la tabulation

Le Langage TC-LOGO

- Exemple de programme TC-LOGO:

```
FORWARD 100 FORWARD 10
REPEAT 10
     [FORWARD 10 FORWARD 100]
```

```
REPEAT 360
[FORWARD 1
LEFT
1]
```

- Grammaire (S,T,N,P),
- S={ PROG }, N={ <PROG>, <INST>}
- T={ FORWARD, LEFT, RIGHT, REPEAT, '[', ']', ENTIER, SEPARATEUR }
- ENTIER et SEPARATEUR sont des lexèmes qui seront identifiés par l'analyse lexicale

Grammaire possible pour TC-LOGO

P={

```
<PROG> ::= <INST> | <PROG> <INST>
<INST> ::= FORWARD ENTIER |
           LEFT ENTIER |
           RIGHT ENTIER |
           REPEAT ENTIER '[' <PROG> ']'
```

}

Lex et Yacc en résumé

- Outils pour produire des parseurs
- Utiles pour traiter les fichiers de données ou pour analyser des formats simples.
- Outils open source (gnu) extrêmement solides et portables (produisent du C travaillant sur les E/S standard).

Table of Contents

Exemple de grammaire ambiguë

- Une grammaire est ambigue lorsque deux arbres de dérivations différents peuvent donner le meme mot du langage.
- Exemple: grammaire ambigue pour *if-then-else*

1	*Instruction*	$\rightarrow$	*if Expr then Instruction else Instruction*
2		$\mid$	*if Expr then Instruction*
3		$\mid$	*Assignation*
4		$\mid$	*AutreInstructions....*

- Avec cette grammaire, le code

$$if\ Expr_1\ then\ if\ Expr_2\ then\ Ass_1\ else\ Ass_2$$

- peut être compris de deux manières:

if $Expr_1$
then if $Expr_2$
then Ass_1
else Ass_2

if $Expr_1$
then if $Expr_2$
then Ass_1
else Ass_2

Solution: desambiguitisation (desambiguation ?)

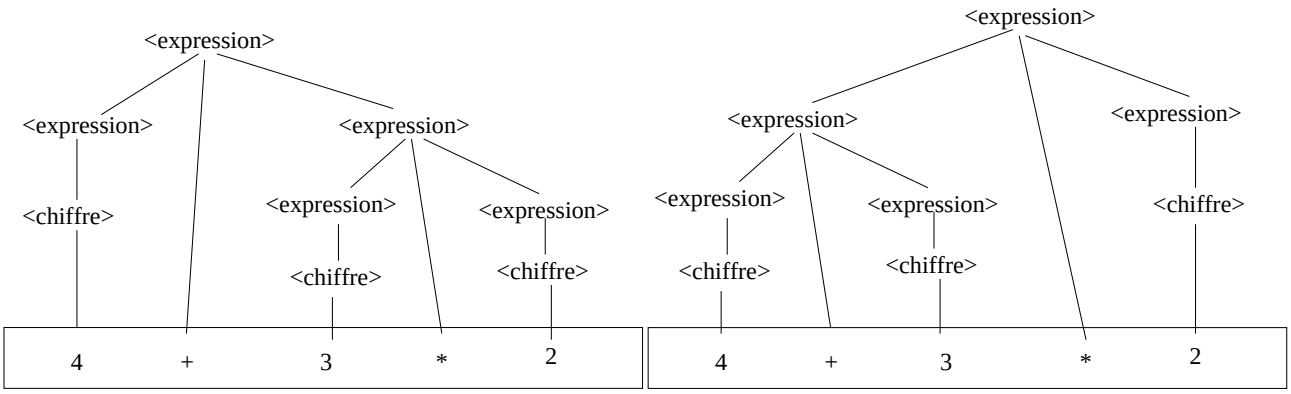
- Il faut changer la grammaire

1	*Instruction*	$\rightarrow$	*AvecElse*
2		$\mid$	*DernierElse*
3	*AvecElse*	$\rightarrow$	*if Expr then AvecElse else AvecElse*
4		$\mid$	*Assignation*
5		$\mid$	*AutreInstructions....*
6	*DernierElse*	$\rightarrow$	*if Expr then Instruction*
7		$\mid$	*if Expr then AvecElse else DernierElse*
8		$\mid$	*AutreInstructions....*

Retour sur la grammaire des expression ambiguë

- Soit la grammaire suivante:

```
<expression> ::= <expression> + <expression> |
                 <expression> * <expression> |
                 <number>
<number> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```



- Deux dérivations sont équivalentes si elles ont le même arbre de dérivation (seul l'ordre dans lequel on a choisit les règles peut changer).

Solution: modifier la grammaire

- ... Sans modifier le langage:

```
<expression> ::= <expression> + <expression> |
                 <terme>
<terme>      ::= <terme> * <terme> | <number>
<number> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

- $\Rightarrow$ un seul arbre de dérivation possible.
- On peut faire d'autres améliorations pour l'associativité des opérateurs et les parenthèses. Une grammaire couramment utilisée pour les expressions arithmétiques est la suivante:

```
<expression> ::= <expression> + <terme> | <terme>
<terme> ::= <terme> * <facteur> | <facteur>
<facteur> ::= ( <expression> ) | <number>
<number> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Sens de la récursion

- On a souvent le choix du sens de la récursion:

$$\left\{ \begin{array}{l} List \rightarrow \underline{elem}\ \ List \\ List \rightarrow \underline{elem} \end{array} \right\} \quad \left\{ \begin{array}{l} List \rightarrow List\ \ \underline{elem} \\ List \rightarrow \underline{elem} \end{array} \right\}$$

à droite ou gauche

- La récursion à gauche est préférable pour des raisons de performances (taille de la pile générée pendant le parsing)
- Mais attention, cela influe sur l'associativité implicite de l'opérateur: si on choisi la récursion à gauche, *elem1 elem2 elem3 elem4* sera interprété comme *(((elem1 elem2) elem3) elem4)* (associatif à gauche).
- La plupart des opérateurs sont associtatifs à gauche.
- Contre exemple: l'affectation en C. `a=b=c` $\Leftrightarrow$ `a=(b=c)`