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

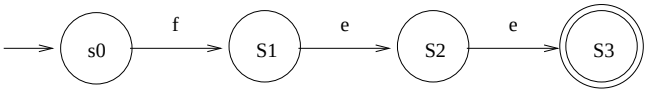
Sources:

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

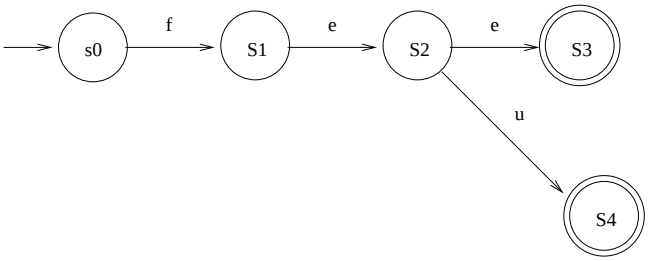
Table of Contents

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`:

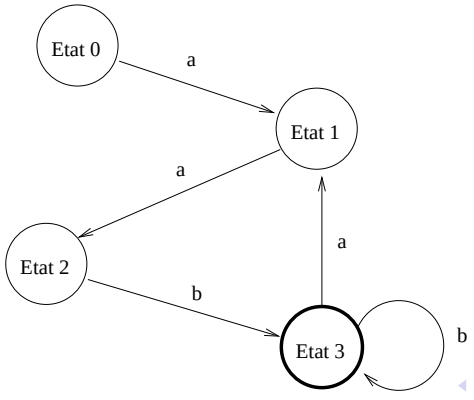


- Automate reconnaissant le langage {`fee`, `feu`}

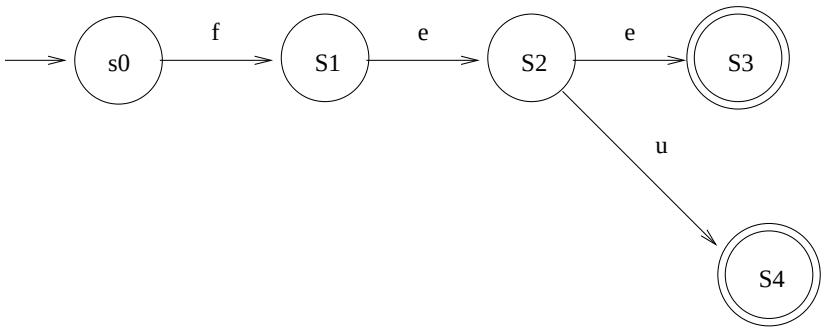


Notion d'automate

- 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

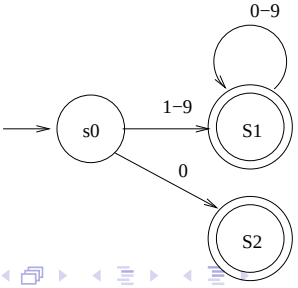


- `fee` → reconnu
- `feu` → reconnu
- `fei` → non reconnu (impossible de lire `'i'`)
- `fe` → non reconnu (arrêt dans un état non final)

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;
  - $\delta : S \times \Sigma \to S$ 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)
  $\delta = \{\delta(s_0, f) \to s_1, \delta(s_1, e) \to s_2, \delta(s_2, e) \to s_3\}$.
  - 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

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 *hift/ educe a ing*), 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 - ight can*, *Reve e- ightmo t de ivation with 1 ymbol lookahead*.
  - La grammaire ne doit pas être ambiguë
  - Lors de chaque réduction, yacc réalise des *action* qui permettent de construire, *en a ant*, 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.

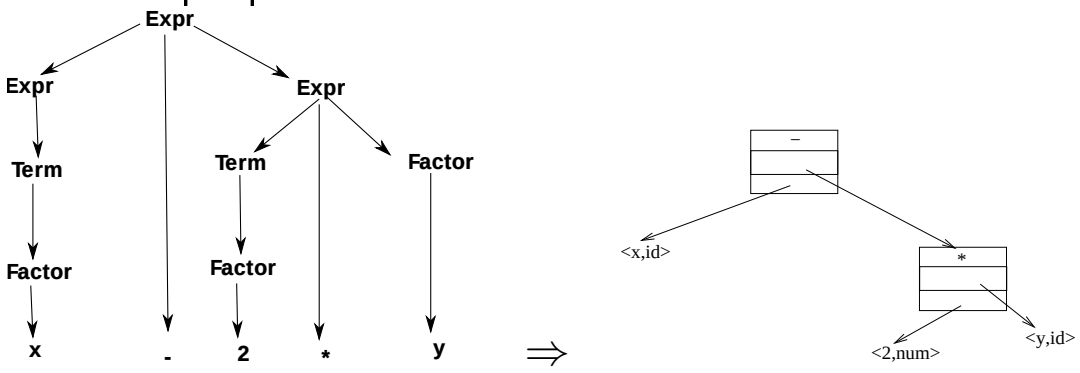


Table of Contents

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

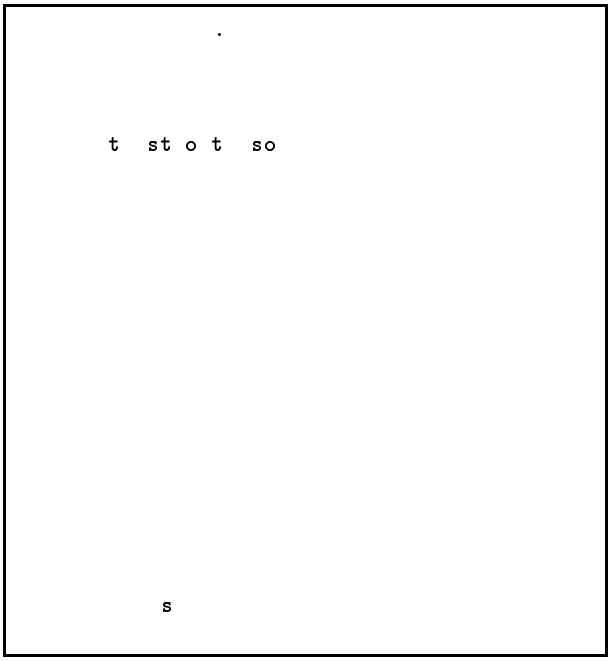
- On doit lire une suite d'addition et afficher le résultat.
  - entrée du parser: 3+4+6
  - affichage du parseur: `résultat: 13`
- Proposition de grammaire élémentaire:
  - Grammaire (S,T,N,P),
  - S={ OM },
  - N={ EXPR, OM }
  - T={ ENTIER, '+' }
  - P={ SOM ::= EXPR
    EXPR ::= NUMBER |
    EXPR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

Additionneur simple: ficher adder.l

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

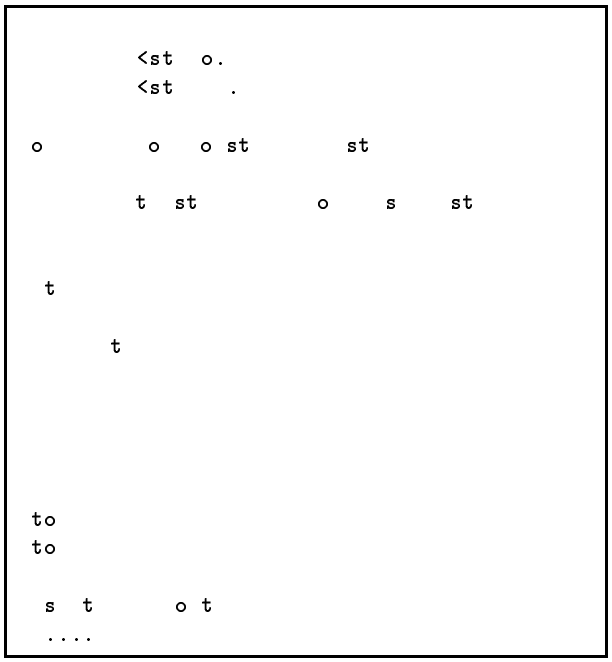
- `adder.tab.h` sera généré par la commande `yacc`, à inclure dans le fichier `lex.yy.c`
- `yyval` et `yytext` sont des variables partagées par `lex` et `yacc`: quand `lex` reconnaît un lexème, il le met dans la variable `yytext` et le transmet à `yacc`. Quand c'est une valeur, `yacc` s'attend à le trouver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

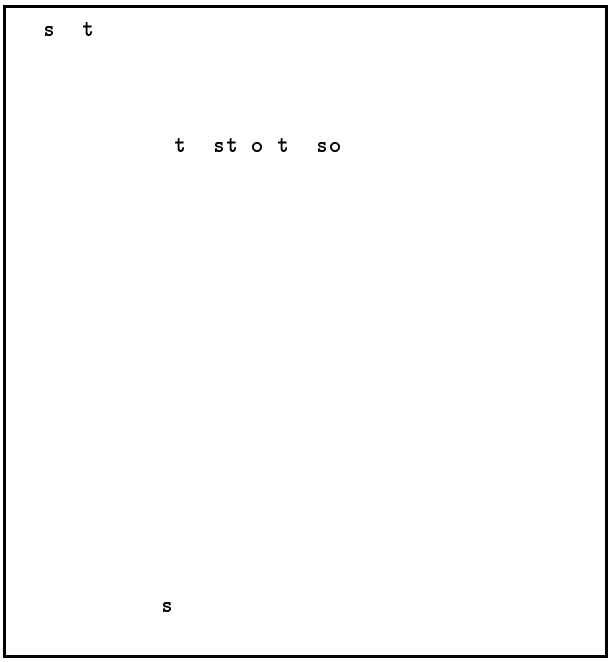
Additionneur simple: grammaire ⇒ adder.y



- 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





- 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

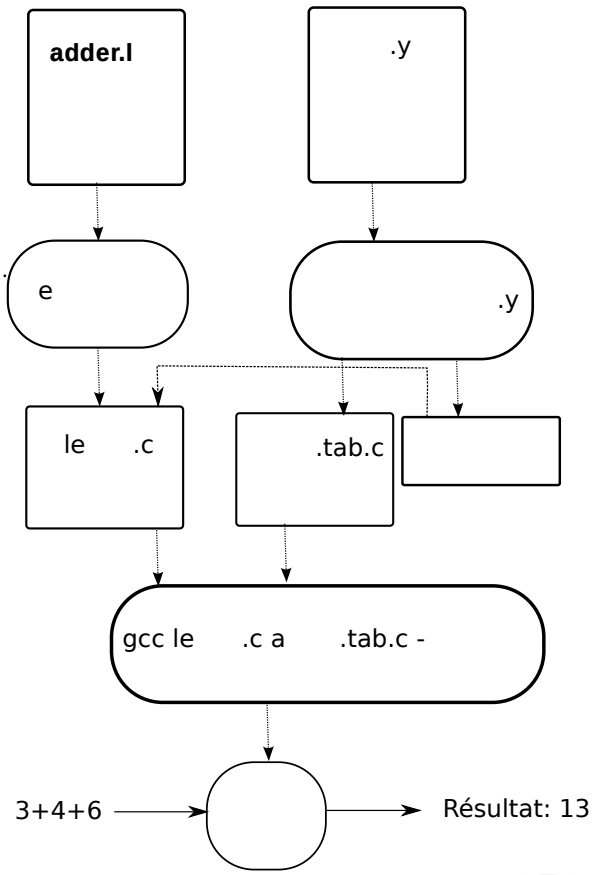
```
...      t o  ...

....     s...

... o  ...
```

- 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 `ain` qui appelle `yyparse` comme ici

Adder: Processus de compilation global



Compilation et utilisation du programme adder

compilation:

```
trisset@fania:~/$  ake
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
so  e:
trisset@fania:~/$ ./adder
1+2
+

+9
^D
so  e: 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:
  - `ORWARD` *n*
  - `LE T` *n*
  - `RI  T` *n*
  - `REPEAT [` *og amme 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 ?)

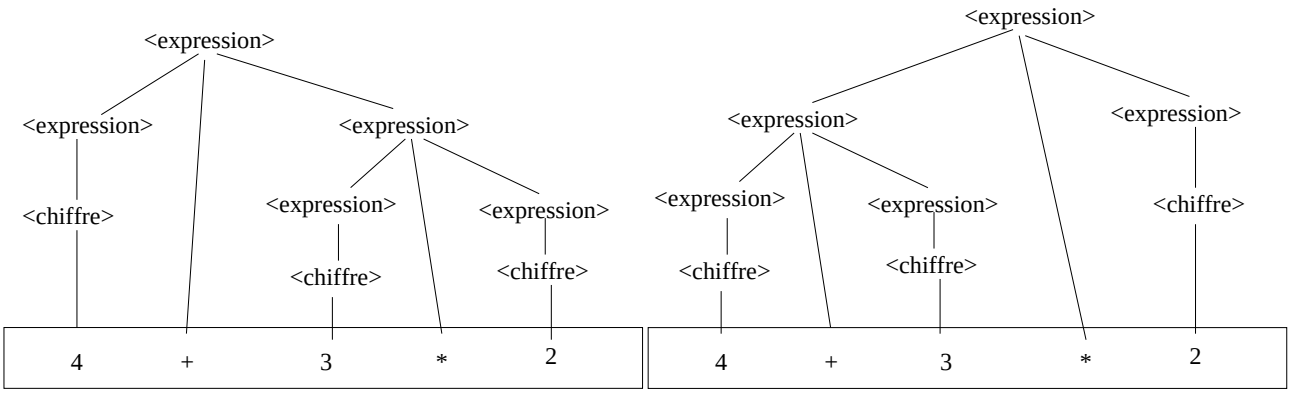
- Il faut changer la grammaire

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)`