Introduction à la Compilation

Nicolas Delestre

INSA ROUEN

v3.1

Architecture des Systèmes d'Information

Note...

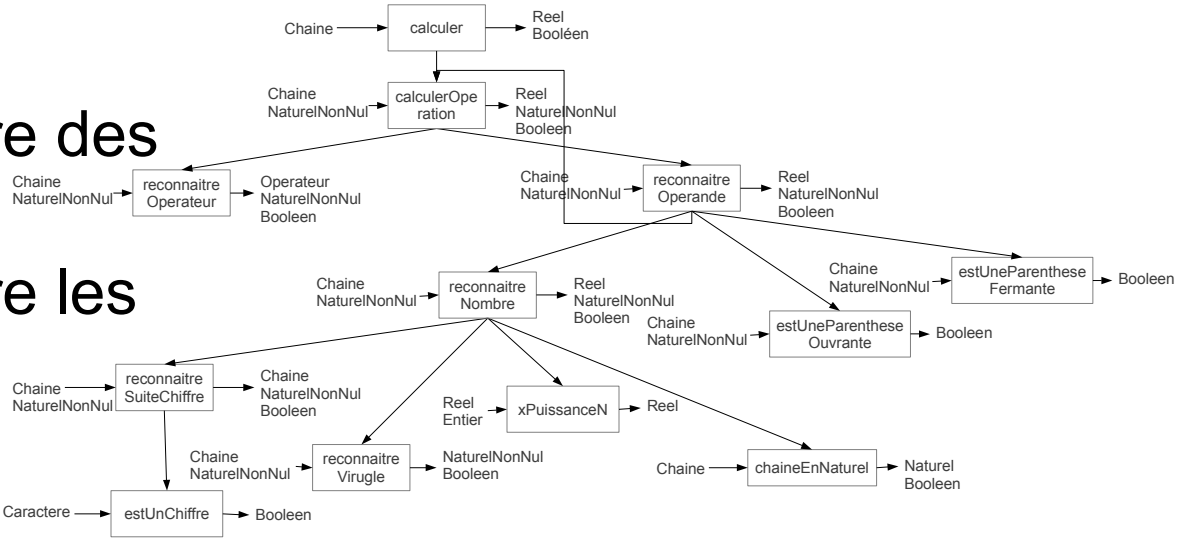
- Ce document est une **introduction** par l'exemple à la compilation
- Il n'est en aucun cas un cours approfondi décrivant les différentes étapes d'un compilateur

Préambule...

- Si vous deviez écrire un programme qui soit capable d'évaluer (au sens de calculer) des expressions arithmétiques qu'un utilisateur aurait saisies au clavier
  - Par exemple : « (24+45.5)/87*1.2E3 »
- Comment procéder ?

Préambule...

- Écrire des procédures et fonctions permettant :
  - De reconnaître des opérandes
  - De reconnaître les opérations
  - …



- C'est compliqué pour quelque chose de simple
- À la moindre modification tout est à reprogrammer

Préambule...

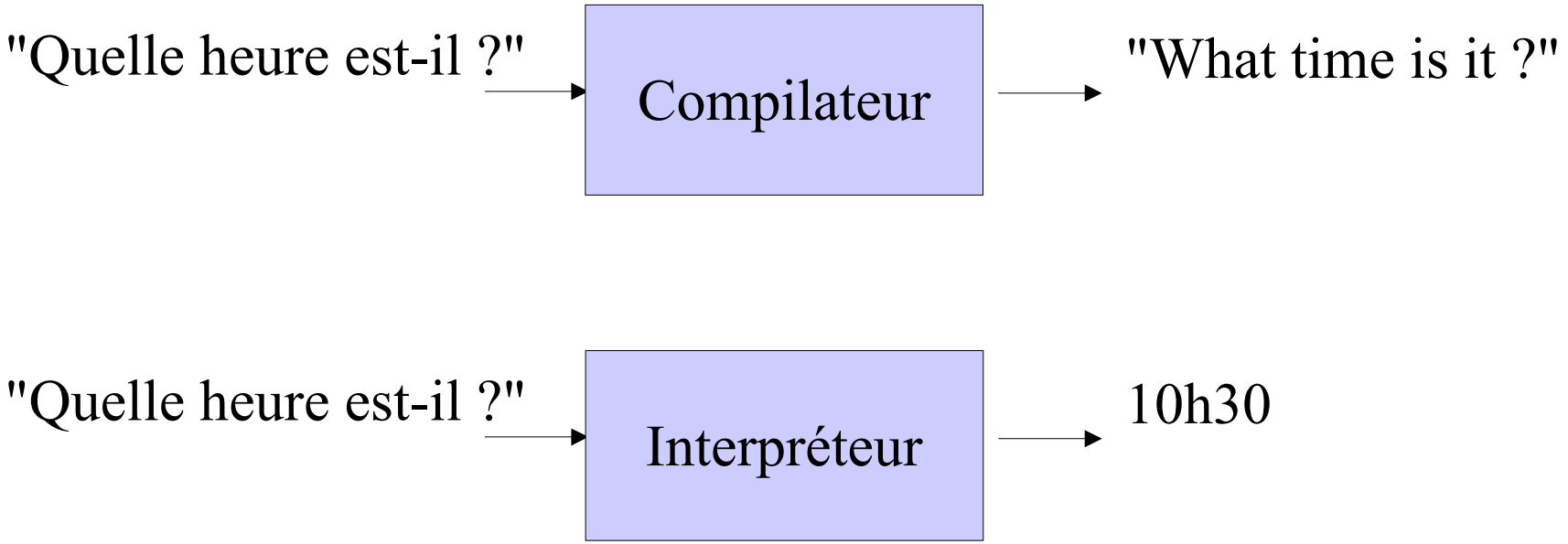
- Il faut suivre des méthodes et techniques afin de résoudre ce type de problème :

C'est le rôle de la

COMPILATION

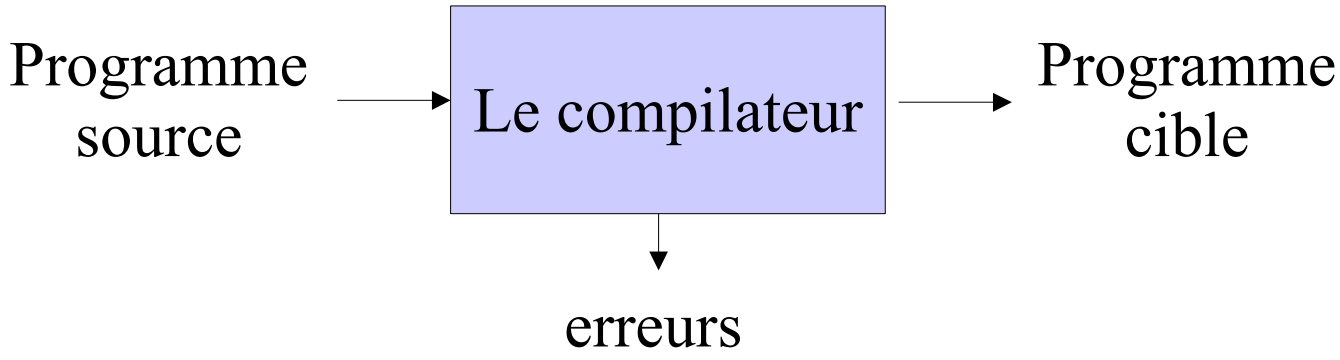
Généralités...

- Compilateur / Interpréteur :



Généralités...

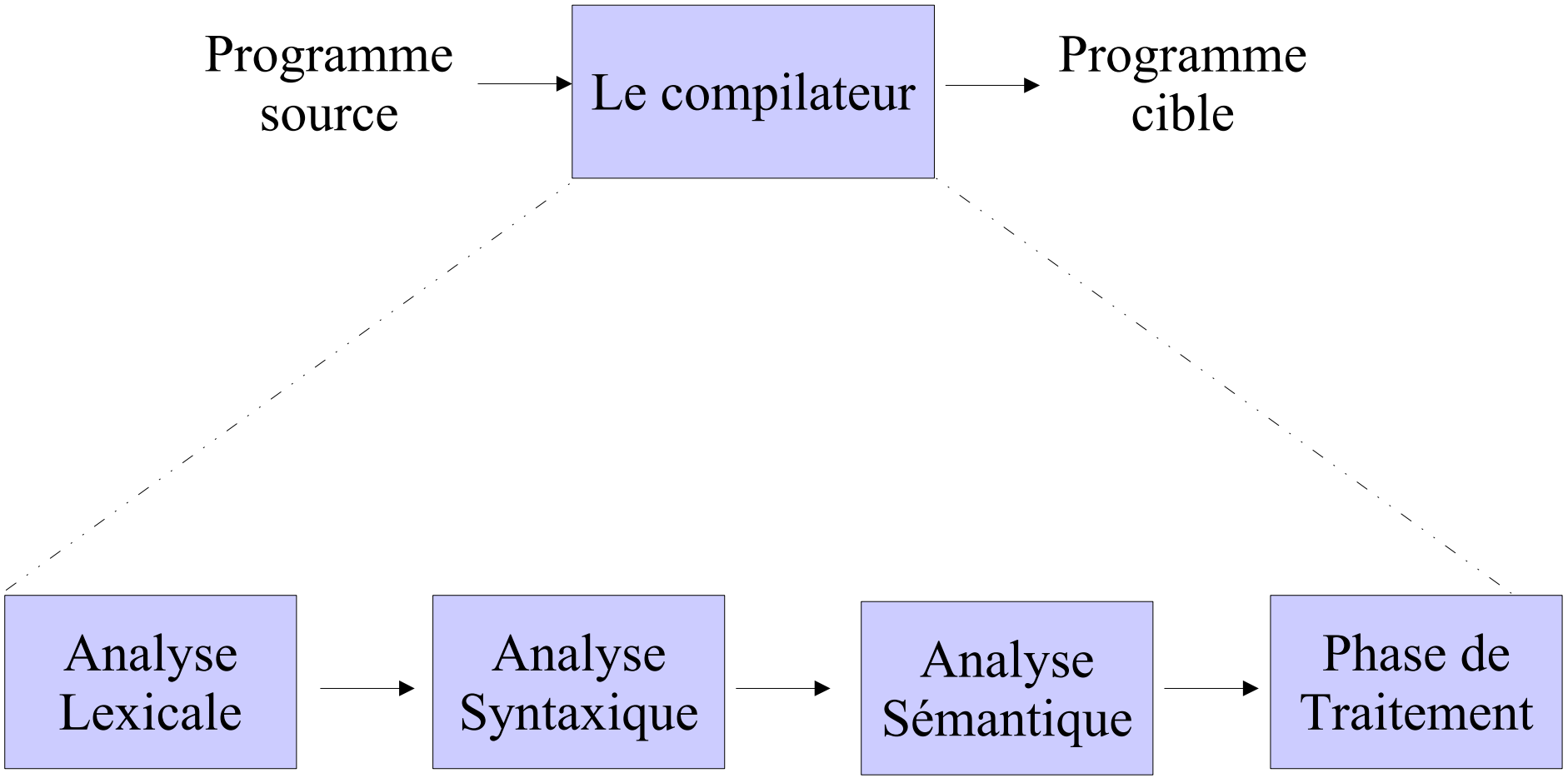
- Qu'est ce qu'un compilateur ?
  - *Définition générale*
    - C'est un programme qui traduit un programme écrit dans un langage source vers un langage cible en indiquant les erreurs éventuelles que pourrait contenir le programme source



Et comment ca marche ?

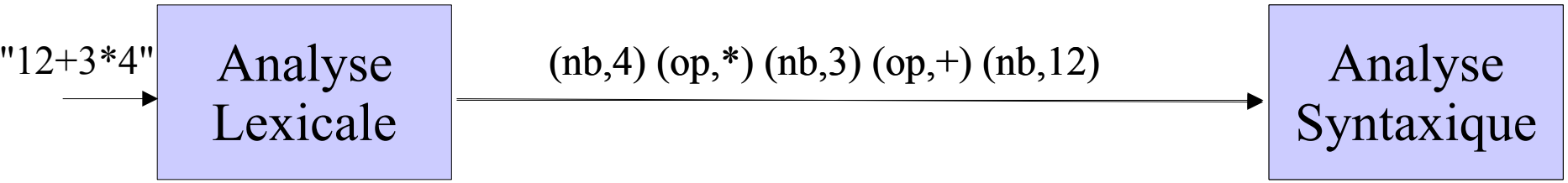
- Comme un enfant qui apprend à lire...
  - On reconnaît d'abord les mots
    - ↳Analyse Lexicale
  - Puis, on vérifie que tous les mots sont dans le bon ordre
    - ↳Analyse Syntaxique
  - Enfin, on vérifie que tout ceci à un sens
    - ↳Analyse Sémantique

Et comment ca marche ?



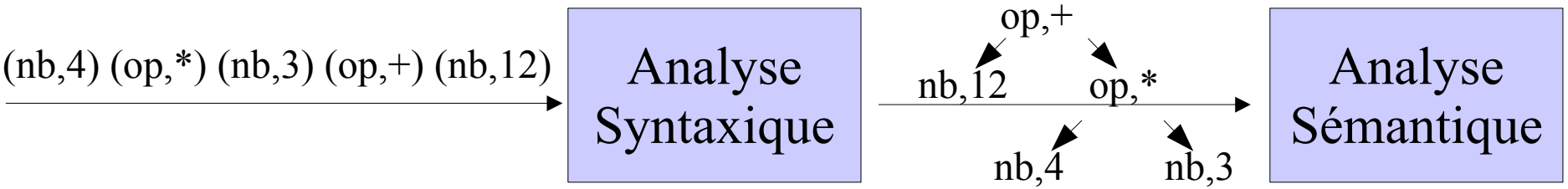
Un exemple : "12+3*4"...

- Analyse lexicale :
  - "12+3*4" est composé des mots :
    - "12" : un nombre dont la valeur est 12
    - "+" : un opérateur binaire dont la valeur est l'addition
    - "3" : un nombre dont la valeur est 3
    - "*" : un opérateur binaire dont la valeur est la multiplication
    - "4" : un nombre dont la valeur est 4
  - Un flot de couples (unité lexicale, valeur) va être passé à l'analyseur syntaxique



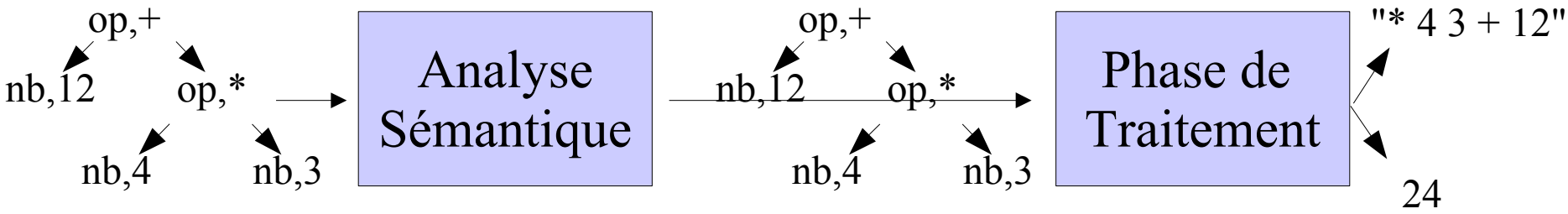
Un exemple : "12+3*4"...

- Analyse syntaxique :
  - Vérifie que la suite des unités lexicales est compatible avec une grammaire et produit en sortie un arbre d'analyse
  - Exemple de règles de la grammaire :
    - Une expression arithmétique est composée d'opérations
    - Une opération est soit unaire soit binaire
    - Une opération binaire est composée d'un opérande, d'un opérateur et d'un opérande
    - L'opérateur * est prioritaire sur l'opérateur +
    - Un opérande est un nombre ou une expression arithmétique ou une variable



Un exemple : "12+3*4"...

- Analyse sémantique :
  - Vérifie que l'arbre d'analyse est cohérent
  - Exemple de règles de cohérence :
    - On ne peut additionner que des nombres
    - On ne peut multiplier que des nombres
    - ...



Analyse lexicale...

```
position := initiale + vitesse * 60
```

- But : `<id,position><aff,><id,initiale><op,add><id,vitesse><op,mul><nb,60>`
  - Flot de caractères

    → flot de couples (unité lexicale, valeur)
- Comment reconnaître les lexèmes :
  - Les lister : avoir un dictionnaire de lexèmes
  - Mais certains mots sont **génériques** :
    - Les nombres
    - Les identifiants des programmes (variable, fonction/procédure, etc.)

      ↳ Il faut un outil permettant de les caractériser

Les expressions rationnelles (ou régulières)

Expressions rationnelles...

- Composition d'opérations :
  - Type : Ensemble des caractères disponibles (nommé Alphabet noté $\sum$)
  - Opérations disponibles :
    - . : opération binaire de concaténation (facultatif)
    - | : opération binaire de réunion
    - * : opération unaire (de 0 à n concaténation)
    - Opérations secondaires
      - + : opération unaire (de 1 à n concaténation)
      - ? : opération uniare (0 ou 1)
- La valeur d'un expression rationnelle est un ensemble de mots (nommé langage dénoté par l'expression rationnelle)

Exemples...

- Avec $\sum$ = {a,b} :
  - a|b dénote {a,b}
  - (a|b)(a|b) dénote {aa,ab,ba,bb}
  - $(a|b)^*$ dénote tous les mots composés de a et de b et le mot vide
  - $a|a^*b$ dénote {a, b, ab, aab, aaab, ...}
- Certains langages ne peuvent être représentés à l'aide d'expressions rationnelles, on parle alors d'ensembles non rationnels (non régulier)
  - Par exemple :
    - ensembles équilibrés ou imbriqués
    - {wcw | w est un mot composé de a et de b}

Définition rationnelle...

- Pour des raisons de commodités, on peut souhaiter nommer les expressions rationnelles
- Une définition rationnelle est une suite de définitions de la forme :

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\ldots$$
$$d_n \rightarrow r_n$$

où d_i est un identifiant distinct et r_i est une expression rationnelle sur $\sum \cup \{d_1, d_2, \ldots, d_{i-1}\}$

Définition rationnelle...

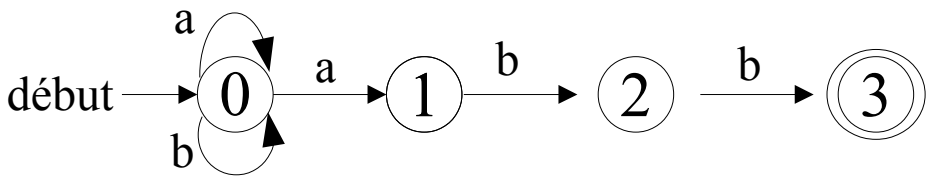
- Exemple :
  - Définition rationnelle des identificateurs :
    - **lettreMaj** $\rightarrow$ A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
    - **lettreMin** $\rightarrow$ a|b|c|d|e|f|g|h|i|j|q|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
    - **chiffre** $\rightarrow$ 0|1|2|3|4|5|6|7|8|9
    - **id** $\rightarrow$ **lettreMaj|lettreMin** (**lettreMaj|lettreMin|chiffre**)*

Exemple complet...

- Nombres non signés en Pascal :
  - **chiffre** $\rightarrow$ 0|1|2|3|4|5|6|7|8|9
  - **chiffres** $\rightarrow$ **chiffre**$^{+}$
  - **decimale_opt** $\rightarrow$ (.**chiffres**)?
  - **exposant_opt** $\rightarrow$ (E(+ | -)? **chiffres**)?
  - **nb** $\rightarrow$ **chiffres decimale_opt exposant_opt**

Comment on implante tout ca ?

- La théorie montre que toute expression rationnelle peut être représentée à l'aide d'un automate à état fini (et inversement)
- Automate à état fini = graphe orienté étiqueté valué (par les lettres de l'alphabet)
  - Exemple :
    - $(a|b)^*abb$

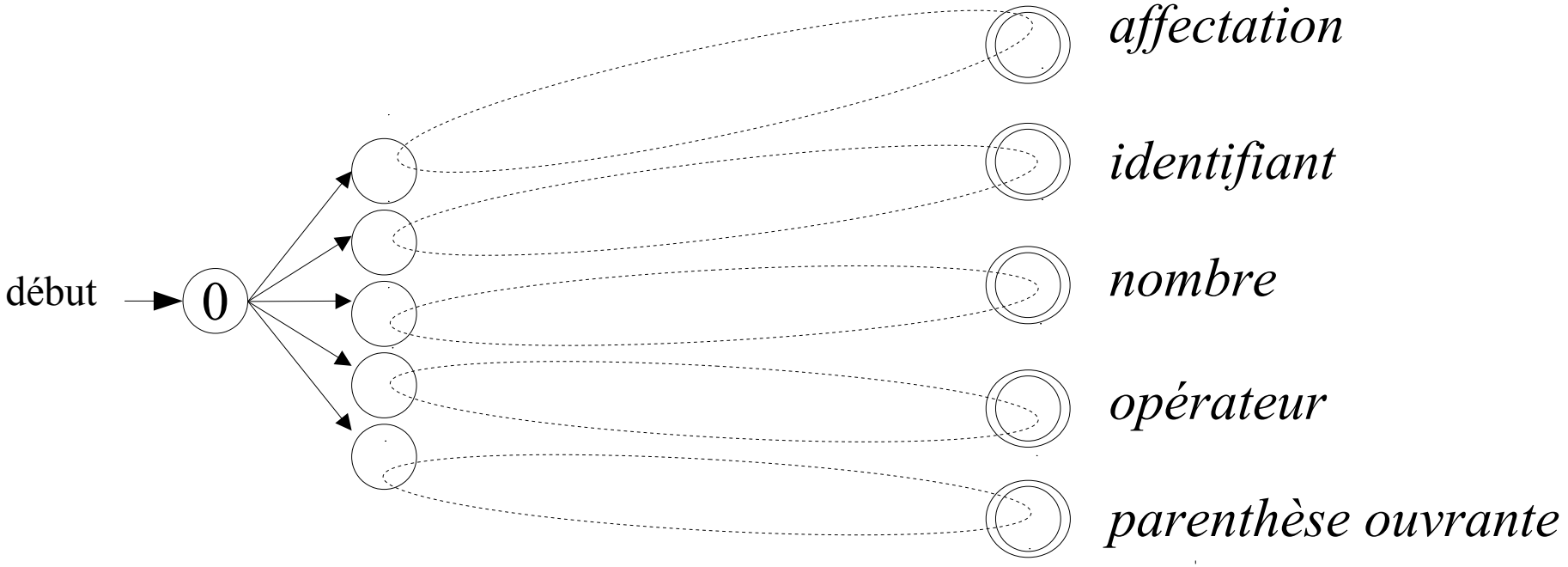


    - ababb mot du langage ?

http://www.logic.at/staff/searles/jautomata/applet.html

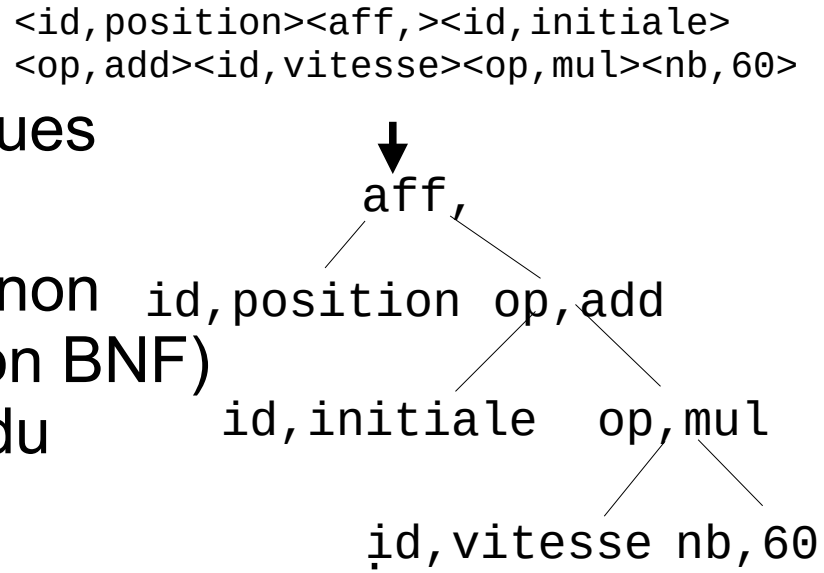
En résumé

- Un analyseur lexical est un ensemble d'expressions rationnelles (et donc autant d'automates) auxquelles on associe une unité lexicale et si besoin une méthode de calcul de valeur



Analyse Syntaxique...

- But :
  - Flot d'unités lexicales → **arbre abstrait**
  - Fonctionnement basé sur les structures hiérarchiques d'un programme
  - Utilisation de grammaires non contextuelles (ou notation BNF) pour décrire la syntaxe du langage
  - Les langages dénotés par une grammaire non contextuel sont appelés langages algébriques
  - Deux types d'analyse : Ascendante ou Descendante



Grammaire...

- Grammaire non contextuelle :
  - Permet d'exprimer des règles du type :

    Si S1 et S2 sont des instructions et E une expression, alors

    « si E alors S1 sinon S2 » est une instruction

    ….de la façon suivante :

    instr $\rightarrow$ **si** *expr* **alors** *instr* **sinon** *instr*
  - Composée :
    - symboles terminaux (les unités lexicales)
    - symboles non-terminaux (gauche des règles)
    - l'axiome (non-terminal particulier, de départ)
    - les productions (les règles)

Grammaire...

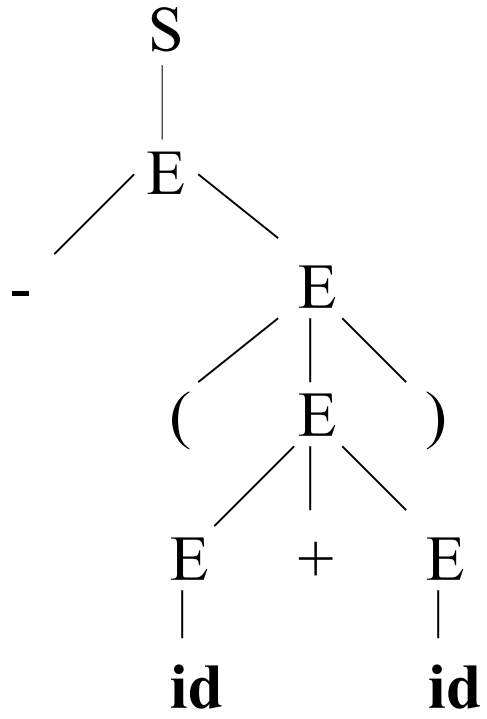
- les conventions de notation :
  - Terminaux : **chaîne en gras**
  - Non-terminaux : lettres majuscules du début de l'alphabet, *noms minuscules en italique*
  - Axiome : souvent noté S
  - Symboles grammaticaux : lettres majuscules de fin de l'alphabet
  - Chaîne de symboles grammaticaux : ‘ α ’, ’β ’,…
  - Utilisation du symbole |
  - Exemple :
    - S $\rightarrow$ E
    - E $\rightarrow$ E O E | **(**E**)** | **-**E | **id** | **nb**
    - O $\rightarrow$ **+** | **-** | ***** | **/** | **^**

Analyse descendante...

- Dérivation
  - Chaîne obtenue en commençant par l'axiome et en remplaçant de manière répétée un non-terminal par la partie droite d'une des productions le définissant
    - $S \Rightarrow E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$
  - Notation :
    - $\alpha A \beta \Rightarrow \alpha\gamma\beta$ (Si il existe $A \rightarrow \gamma$)
    - Dérivation en 0 ou plusieurs étapes : $\Rightarrow^*$
    - Dérivation en 1 ou plusieurs étapes : $\Rightarrow^+$
    - Dérivation gauche : $\Rightarrow_g$
    - Dérivation droite : $\Rightarrow_d$

Analyse descendante...

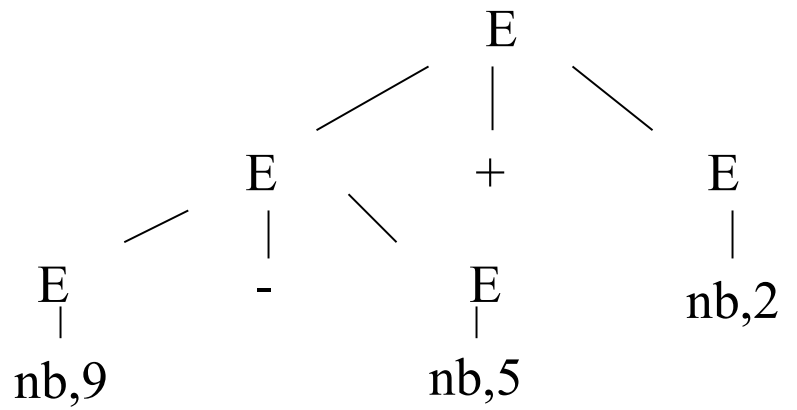
- Une dérivation peut alors donner un arbre d'analyse : S $\Rightarrow$ E $\Rightarrow$ -E $\Rightarrow$ -(E) $\Rightarrow$ -(E+E) $\Rightarrow$ -(**id**+E) $\Rightarrow$ -(**id**+ **id**)

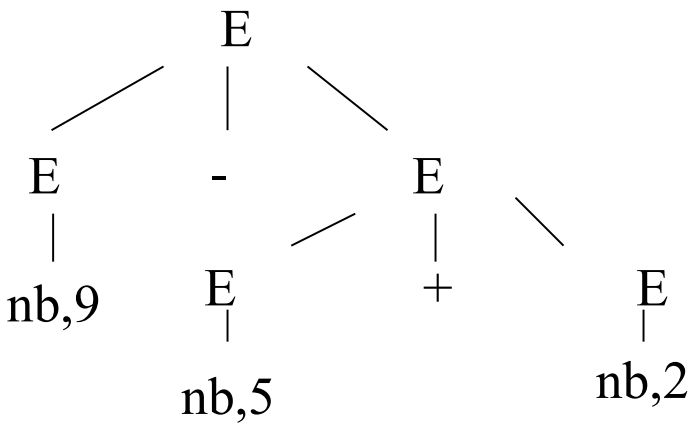


Grammaire ambiguë...

- Un grammaire est dit ambiguë lorsque deux arbres différents peuvent être obtenus à partir d'une même suite d'unité lexicale
- Exemple : E $\rightarrow$ E**+**E | E**-**E | **nb**

9-5+2



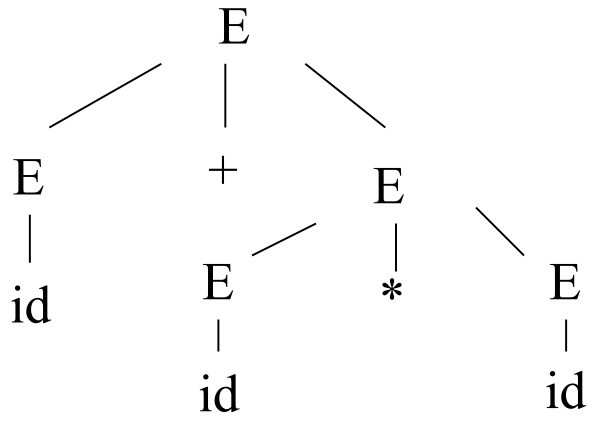


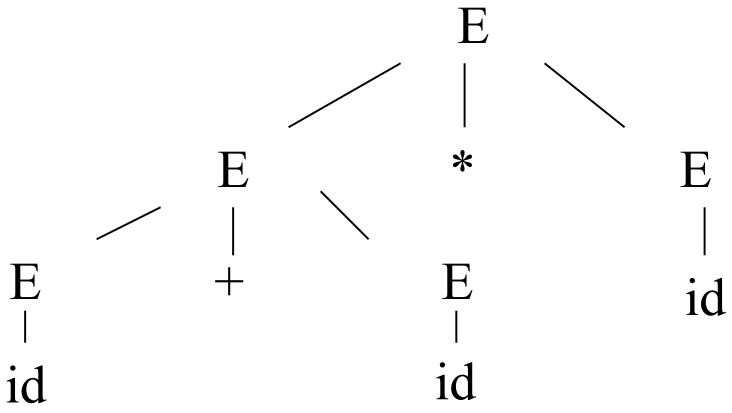
Grammaire ambiguë...

- E $\rightarrow$ E**+**E | E**-**E | E*****E | E/E | **(**E**)** | **-**E | **id**

  Deux dérivations sont possibles pour obtenir **id+id*id**

$$
\begin{aligned}
E &\Rightarrow E + E \\
&\Rightarrow \mathbf{id} + E \\
&\Rightarrow \mathbf{id} + E * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{aligned}
\qquad
\begin{aligned}
E &\Rightarrow E * E \\
&\Rightarrow E + E * E \\
&\Rightarrow \mathbf{id} + E * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * E \\
&\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{aligned}
$$





Suppression des ambiguïtés...

- Modifier la grammaire :
  - E $\rightarrow$ E **+**T | E**-**T | T
  - T $\rightarrow$ T ***** F | T **/** F | F
  - F $\rightarrow$ **(**E**)** | **-**E | **id**
- E$\Rightarrow$E+T$\Rightarrow$T+T$\Rightarrow$F+T$\Rightarrow$id+T$\Rightarrow$id+T*F$\Rightarrow$ id+F*F$\Rightarrow$id+id*F$\Rightarrow$id+id*T$\Rightarrow$id+id*id
- Il n'y a pas de technique automatique....

Suppression des ambiguïtés...

- Associativité des opérateurs :
  - Exemple :
    - Comment comprendre : 9+5+2 ou a=b=2 ?
  - Associativité à gauche :
    - Un opérande entouré par un même opérateur sera traité par celui de gauche (c'est le cas de +,-,*,/)

      9+5+2 → **9+5+**2
  - Associativité à droite :
    - Un opérande entouré par un même opérateur sera traité par celui de droite (c'est le cas de l'affectation)

      a=b=2 → a=**b=2**
- Priorité des opérateurs :
  - * à une priorité supérieure à +

    ↳ signifie que * agit sur ses opérandes avant +

Analyse ascendante...

Analyse décalage-réduction

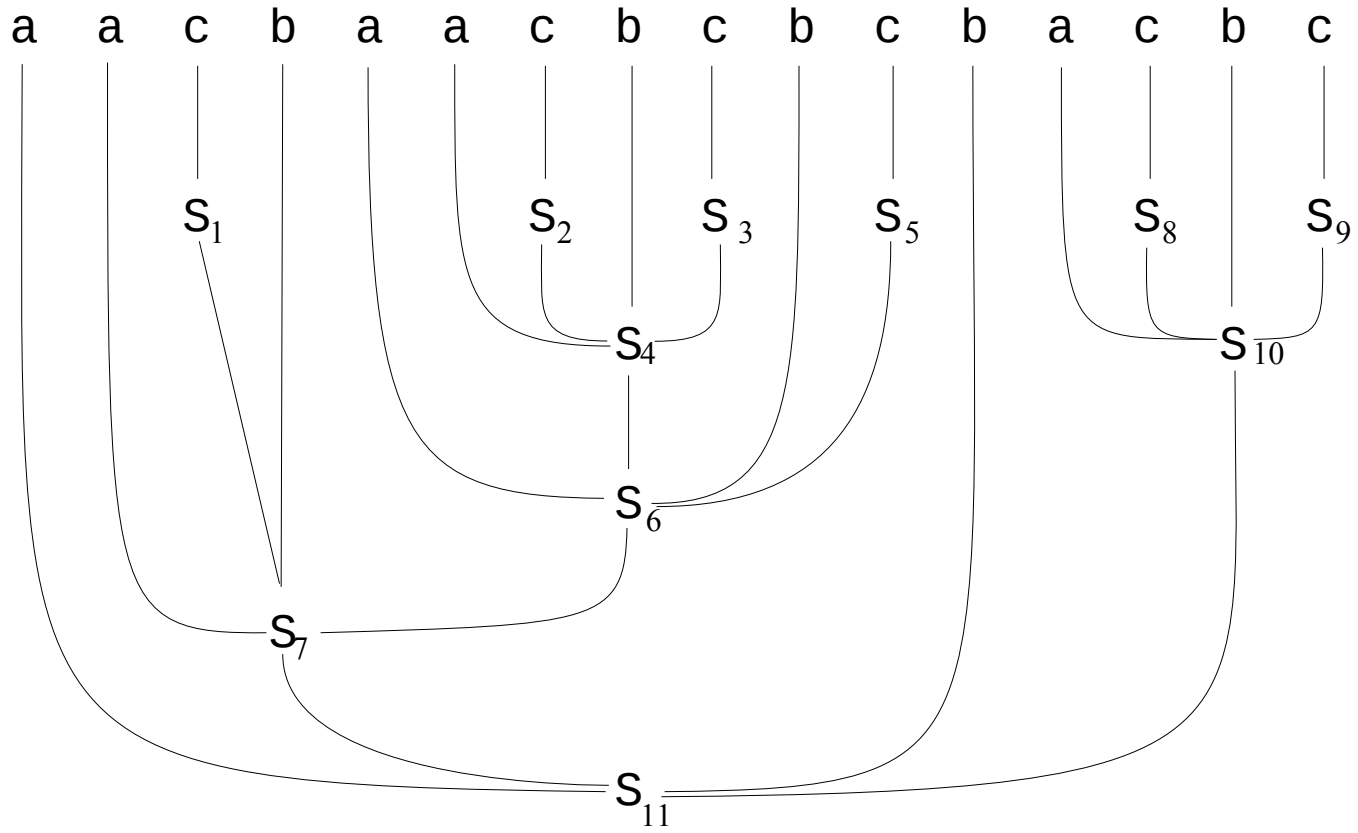
- But :
  - Construire un arbre d'analyse pour une chaîne source en commençant par les feuilles :
    - Peut être considérée comme la réduction d'un chaîne w vers l'axiome de la grammaire
    - A chaque étape de réduction, une sous-chaîne particulière correspondant à la partie droite d'une production est remplacée par le symbole de la partie gauche

Exemple...

S → **a**S**b**S | **c** avec le mot u=aacbaacbcbcbacbc

1	**a**acbaacbcbcbacbc	on ne peut rien **réduire**, donc on **décale**
2	a**a**cbaacbcbcbacbc	on ne peut rien réduire, donc on décale
3	aa**c**baacbcbcbacbc	on peut réduire S → c
4	aa**S**baacbcbcbacbc	on ne peut rien réduire, donc on décale
5	aaS**b**aacbcbcbacbc	on ne peut rien réduire, donc on décale
6	aaSb**a**acbcbcbacbc	on ne peut rien réduire, donc on décale
7	aaSba**a**cbcbcbacbc	on ne peut rien réduire, donc on décale
8	aaSbaa**c**bcbcbacbc	on peut réduire S → c
9	aaSbaa**S**bcbcbacbc	on ne peut rien réduire, donc on décale
10	aaSbaaS**b**cbcbacbc	on ne peut rien réduire, donc on décale
11	aaSbaaSb**c**bcbacbc	on peut réduire S → c
12	aaSbaaSb**S**bcbacbc	on peut réduire S → aSbS
13	aaSba**S**bcbacbc	on ne peut rien réduire, donc on décale
14	aaSbaS**b**cbacbc	on ne peut rien réduire, donc on décale
15	aaSbaSb**c**bacbc	on peut réduire S → c
16	aaSbaSb**S**bacbc	on peut réduire S → aSbS
17	aaSb**S**bacbc	on peut réduire S → aSbS
18	a**S**bacbc	on ne peut rien réduire, donc on décale
19	aS**b**acbc	on ne peut rien réduire, donc on décale
20	aSb**a**cbc	on ne peut rien réduire, donc on décale
21	aSba**c**bc	on peut réduire S → c
22	aSba**S**bc	on ne peut rien réduire, donc on décale
23	aSbaS**b**c	on ne peut rien réduire, donc on décale
24	aSbaSb**c**	on ne peut rien réduire, donc on décale
25	aSbaSb**c**	on peut réduire S → c
26	aSbaSb**S**	on peut réduire S → aSbS
27	aSb**S**	on peut réduire S → aSbS
28	**S**	TERMINE

Exemple...

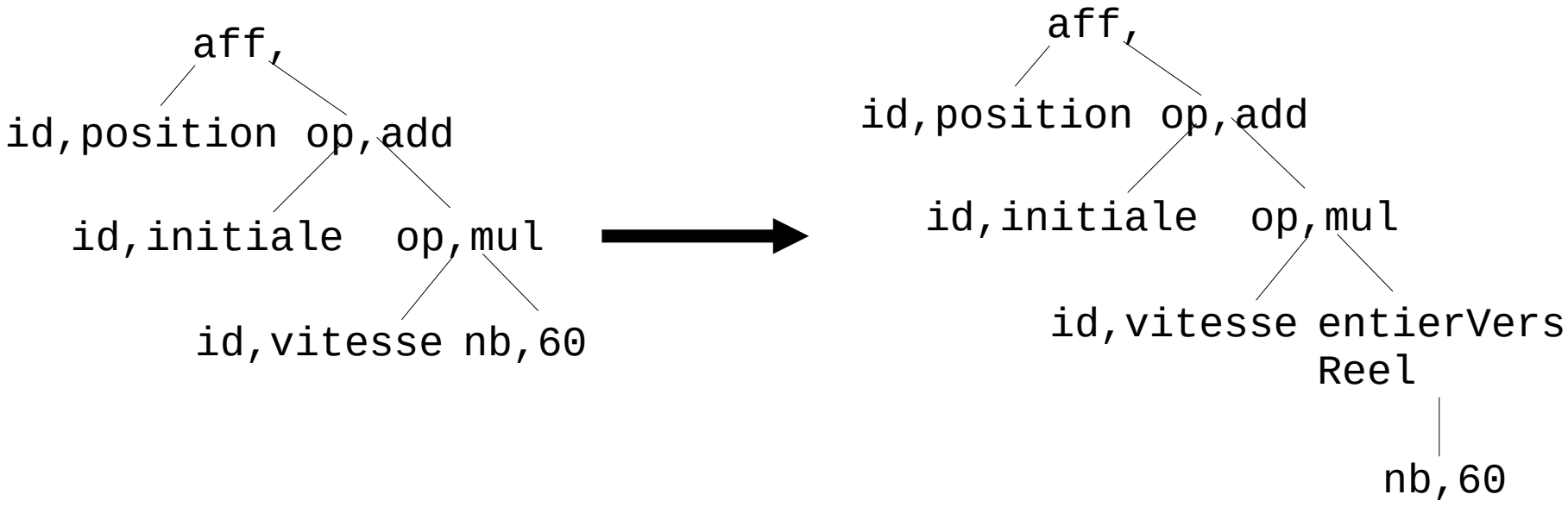


Comment on implante tout ca ?

- La théorie montre que toute grammaire non contextuelle peut être représentée à l'aide d'un automate à pile
- Un automate à pile est un automate à état fini auquel est associé une mémoire organisée sous forme de pile
- Les actions sur un automate à pile peuvent être la transition d'un nœud vers un autre nœud ou l'empilement ou le dépilement d'une information
- L'action à effectuer est fonction de l'unité lexicale courante, du nœud courant et du sommet de la pile

Analyse sémantique...

- Collecte d'informations destinées à la production de code
- Contrôle de type

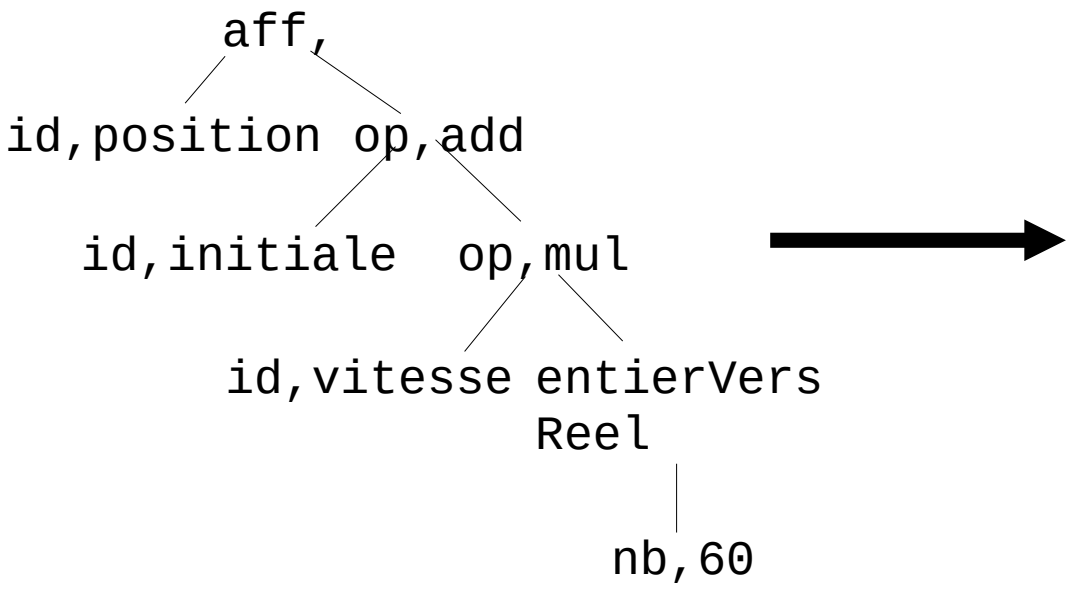


Phase de traitement

- Dépend de l'objectif final :
  - Interprétation ou Compilation
  - Par exemple pour la compilation de code source en assembleur il va y avoir :
    - Générateur de code intermédiaire
    - Optimiseur de code
    - Génération du code cible (très souvent l'assembleur)

Générateur de code intermédiaire...

- Programme pour une machine abstraite
  - Code facile à produire, à traduire en langage cible




```
temp1:=EntierVersReel(60)
temp2:=id3*temp1
temp3:=id2+temp2
id1:=temp3
```

Optimiseur de code...

- Optimiser le code intermédiaire pour une exécution machine plus rapide

```
temp1:=EntierVersReel(60)
temp2:=id3*temp1
temp3:=id2+temp2
id1:=temp3
```

```
temp1:=id3*60.0
id1:=id2+temp1
```

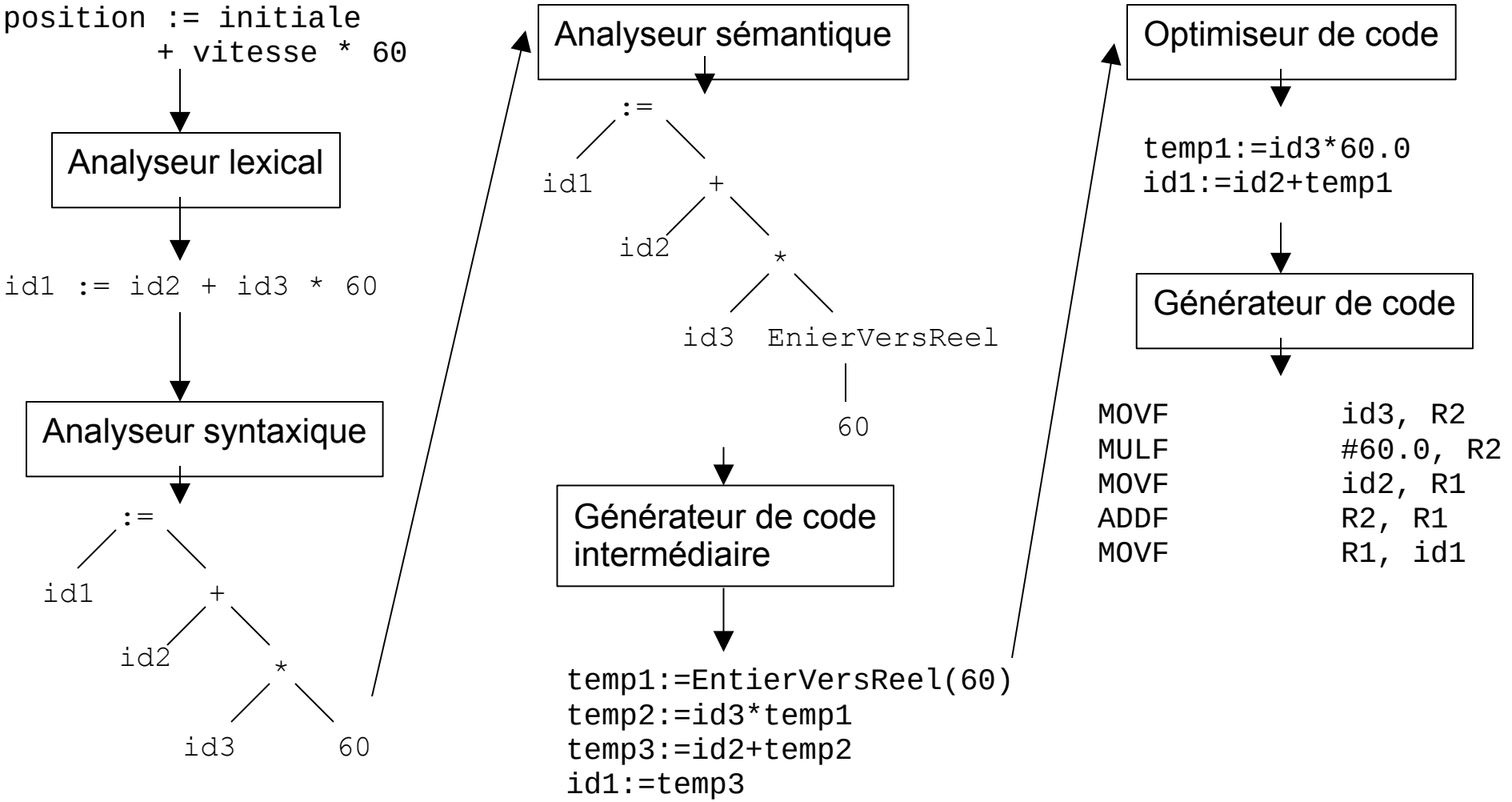
Générateur de code...

- Obtention du code machine

```
temp1:=id3*60.0
id1:=id2+temp1
```

```
MOVF        id3, R2
MULF        #60.0, R2
MOVF        id2, R1
ADDF        R2, R1
MOVF        R1, id1
```

Récapitulatif (notation légèrement différente)

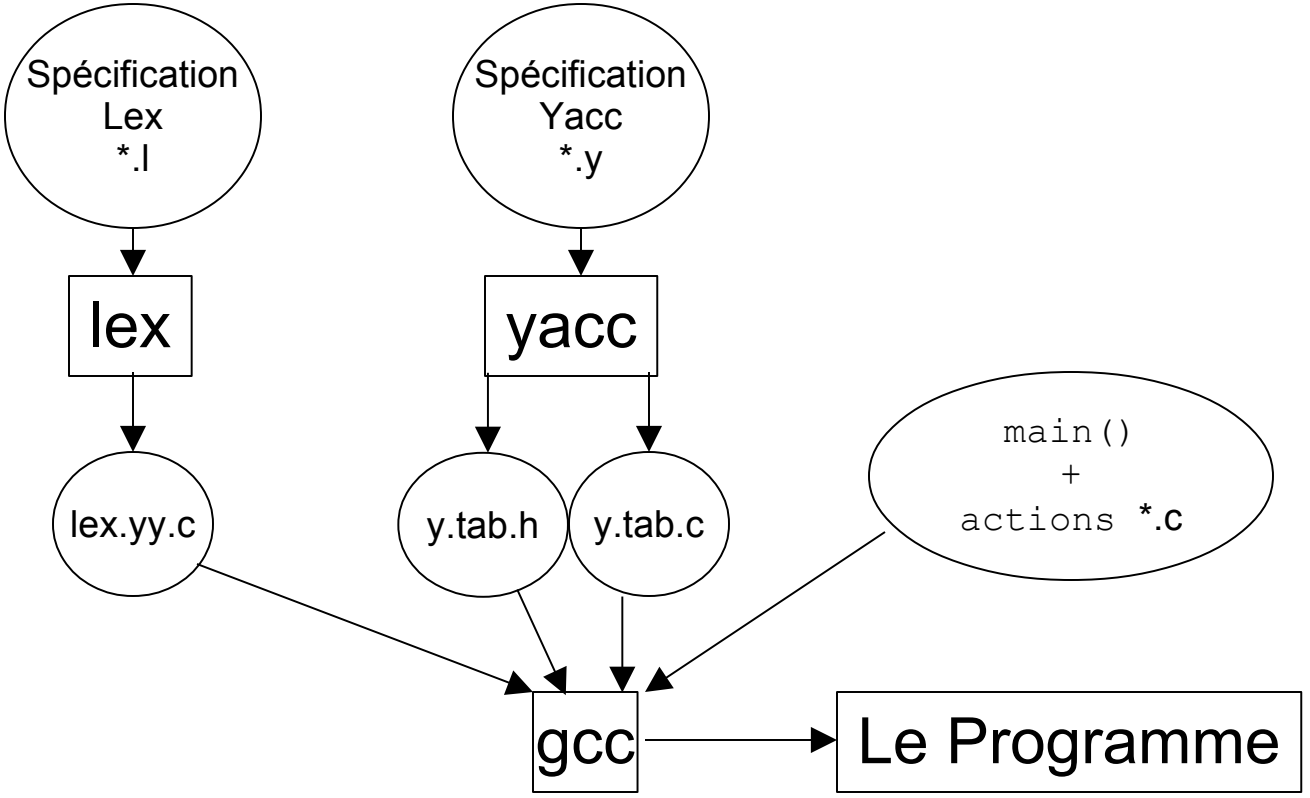


Lex et Yacc...

- Outils permettant de créer facilement des interpréteurs/compilateurs en C
- Lex :
  - Un générateur d'analyseur lexical
  - Génère entre autres une fonction C *yylex()*
- Yacc (Yet Another Compiler of Compiler)
  - Un générateur d'analyseur syntaxique
  - Génère entre autres une fonction C *yyparse()*

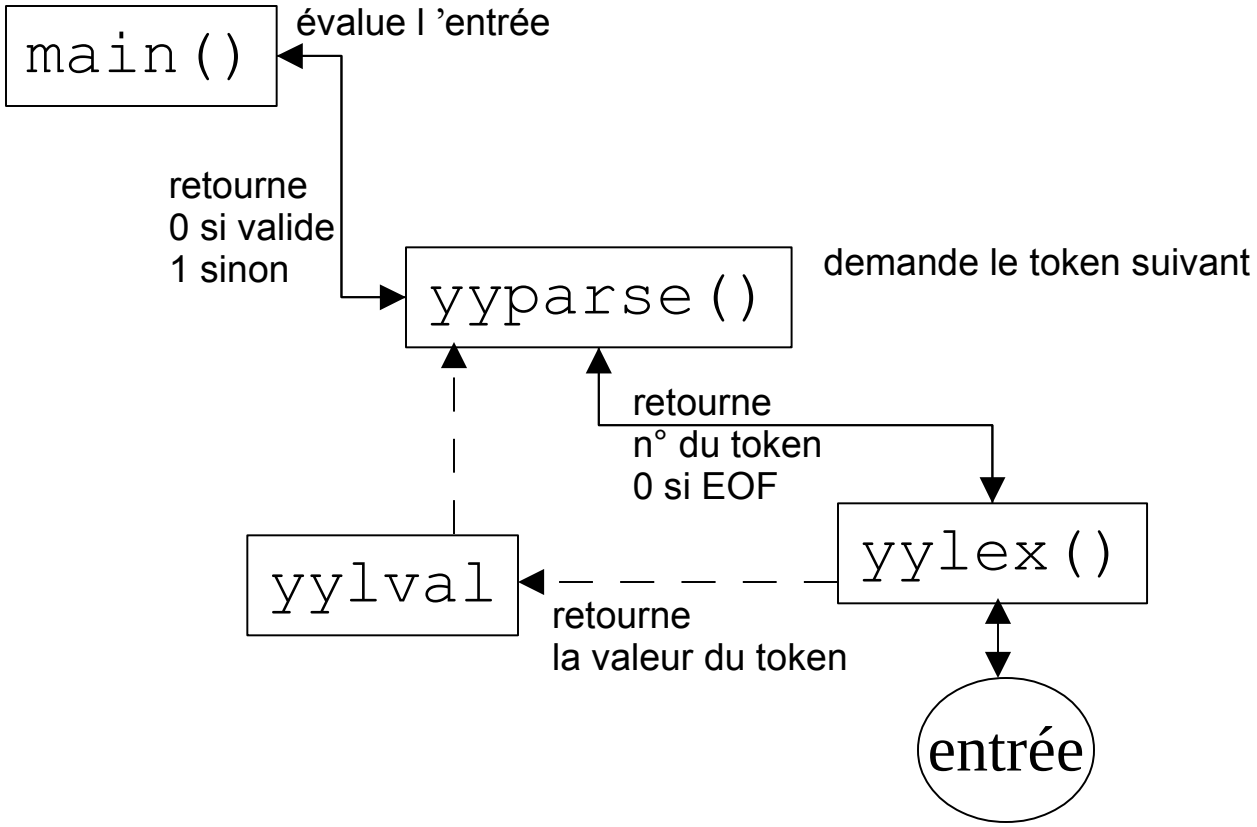
Lex et Yacc...

- Utilisation de Lex et Yacc



Lex et Yacc...

- Collaboration de Lex et Yacc



Lex : structure du fichier source...

```
%{ déclarations C
%}
définitions :
<identificateurs>     <expression rationnelle>
...

%%

règles : <expression rationnelle> <commande en C>

%%
<code C>
```

Lex : variables et fonctions prédéfinies...

- Le programme C produit par Lex contient un certain nombre de variables et de fonctions prédéfinies que l'on peut utiliser :
  - `int yylex()` invoque le lexer et retourne le type du prochain lexème
  - `char* yytext` mot courant
  - `yyleng` longueur du lexème courant
  - `yylval` valeur associé au lexème (type définit dans y.tab.h)
  - `FILE* yyout` fichier de sortie
  - `FILE* yyin` fichier d'entrée

Yacc : Structure du fichier source...

```
%{ déclarations C
%}
définitions
%%

//productions de la grammaire
production    : liste de symboles {actions code C}
              |liste de symboles {actions code C}
              ...
              ;

%%
<programme principal et sous-programmes>
```

Lex et Yacc à partir d'un exemple...

- Une calculatrice à mémoire

```
$ bin/calc
> 2*(3.5+4)
15.000000
> let a=2
> a
2.000000
> 2*a
4.000000
> let b=2*a
> b
4.000000
> 2*a*b
16.000000
> exit
$
```

Lex et Yacc à partir d'un exemple...

- Objectif : Calculatrice (donc évaluateur) à mémoire
  - Analyse lexicale :
    - mots clefs : let, exit, =,(,), +, -, *, /, \n
    - mots génériques :
      - identifiant
      - nombre réel
  - Analyse syntaxique :

$S \rightarrow L$ **\n**

$L \rightarrow C \mid A \mid X$

$A \rightarrow$ **let ID** $= C$

$X \rightarrow$ **exit**

$C \rightarrow$ **ID** | **NB** | **(**C**)** | C+C | C-C | C*C | C/C

Librairies annexes...

- Memoire.h

```
#ifndef __MEMOIRE_H__
#define __MEMOIRE_H__

#include "ListeChaineeDeCellulesMemoire.h"

typedef struct {
  LCDCM_ListeChainee laMemoire;
} Memoire;

Memoire memoire();
void ajouterVariable(Memoire*,char*,double);
void remplacerValeur(Memoire*,char*,double);
int variablePresente(Memoire,char*);
double valeurVariable(Memoire,char*);

#endif
```

Librairies annexes...

- ListeChaineeDeCellulesMemoire.h

```
#ifndef __LISTE_CHAINEE_DE_CELLULES_MEMOIRE__
#define __LISTE_CHAINEE_DE_CELLULES_MEMOIRE__

#include "CelluleMemoire.h"

/* Partie privee */
typedef struct LCDCM_Noeud* LCDCM_ListeChainee;
typedef struct LCDCM_Noeud {
CelluleMemoire lElement;
LCDCM_ListeChainee listeSuivante;
}  LCDCM_Noeud;

/* Partie publique */
LCDCM_ListeChainee LCDCM_listeChainee();
int LCDCM_estVide(LCDCM_ListeChainee);
void LCDCM_ajouter(LCDCM_ListeChainee*,CelluleMemoire);
CelluleMemoire LCDCM_obtenirElement(LCDCM_ListeChainee);
LCDCM_ListeChainee LCDCM_obtenirListeSuivante(LCDCM_ListeChainee);
void LCDCM_fixerListeSuivante(LCDCM_ListeChainee*, LCDCM_ListeChainee);
void LCDCM_fixerElement(LCDCM_ListeChainee*,CelluleMemoire);
void LCDCM_supprimerTete(LCDCM_ListeChainee*);
#endif
```

Librairies annexes...

- CelluleMemoire.h

```
#ifndef __CELLULE_MEMOIRE__
#define __CELLULE_MEMOIRE__

typedef struct {
  char* nomVariable;
  double valeur;
} CelluleMemoire;

CelluleMemoire celluleMemoire(char*,double);
char* obtenirNomVariable(CelluleMemoire);
double obtenirValeurVariable(CelluleMemoire);
void fixerValeurVariable(CelluleMemoire*,double);

#endif
```

Lex : exemple...

```
%{
/*
** Base sur l'exemple de Hugues Cassé
** http://www.irit.fr/ACTIVITES/EQ_HECTOR/casse/alcatel/
*/

#include <string.h>
#include <stdlib.h>
#include <math.h>
#include "y.tab.h"
char* dupliquerChaine(char *);
%}

ENT      [0-9]+
EXP      [+-][eE]{ENT}
VIRG     \.{ENT}
NBRE     {ENT}|{ENT}{VIRG}|{ENT}{EXP}|{ENT}{VIRG}{EXP}
IDENT    [_a-zA-Z][_0-9a-zA-Z]*
%%
```

Lex : exemple...

```
[ \t]             ;
"let"             return LET;
"exit"            return EXIT;
{IDENT}           {yylval.ch = dupliquerChaine(yytext); return ID;}
{NBRE}            {yylval.val = atof(yytext); return NB;}
[+]               return OP_ADD;
[-]               return OP_SOUS;
[*]               return OP_MUL;
[/]               return OP_DIV;
[(]               return PAR_G;
[)]               return PAR_D;
[=]               return AFFEC;
[\n]              return FIN;
%%
char* dupliquerChaine(char *chaine) {
  char* nouvelleChaine = malloc(strlen(chaine)+1);
  strcpy(nouvelleChaine, chaine);
  return nouvelleChaine;
}

int yywrap(void){
  return 1;
}
```

Yacc : exemple...

```
%{
/*
** Base sur l'exemple de Hugues Cassé
** http://www.irit.fr/ACTIVITES/EQ_HECTOR/casse/alcatel/
*/
#include <stdio.h>
#include <math.h>
#include "Memoire.h"

#define RIEN 0
#define AFFICHER 1
#define QUITTER 2
%}
%union {double val;char *ch;}      // Definit le type de yylval (nomme
                                   // YYSTYPE)
//Definition des lexemes
%token <val> NB           // Indique le champ de YYSTYPE qui doit etre utilise dans
%token <ch> ID            // l'interpretation des regles de la grammaire
%token LET EXIT
%token PAR_G PAR_D
%token AFFEC FIN
%token OP_ADD OP_SOUS OP_DIV OP_MUL OP_POW
```

Yacc : exemple...

```
// Definition l'ordre des priorites ainsi que l'associativite
%left OP_ADD OP_SOUS
%left OP_MUL OP_DIV
// Designation du champ de YYSTYPE utilise pour les non terminaux
%type <val> calcul
// les parametres de yyparse
%parse-param {Memoire* pMemoire}
%parse-param {double* pResultat}
%parse-param {int* pCommande}
```

Yacc : exemple...

```
%%
commande: ligne FIN {YYACCEPT;}
;

ligne:   calcul              {*pResultat=$1;*pCommande=AFFICHER;}
|LET ID AFFEC calcul         {if (!variablePresente(*pMemoire,$2))
                                      ajouterVariable(pMemoire,$2,$4);
                             else
                                      remplacerValeur(pMemoire,$2,$4);
                             *pCommande=RIEN;
                             }
|EXIT                        {*pCommande=QUITTER;}
;

calcul:  ID                  {$$ = valeurVariable(*pMemoire,$1);}
|NB                          {$$ = $1;}
|PAR_G calcul PAR_D          {$$ = $2;}
|OP_SOUS calcul              {$$ = - $2;}
|calcul OP_ADD calcul        {$$ = $1 + $3;}
|calcul OP_SOUS calcul       {$$ = $1 - $3;}
|calcul OP_MUL calcul        {$$ = $1 * $3;}
|calcul OP_DIV calcul        {$$ = $1 / $3;}
;
```

Yacc : exemple...

```
%%
yyerror(char *msg){printf("Erreur : %s\n",msg);}

int main(void){
  int commande;
  double valeurCalculee;
  Memoire laMemoire=memoire();
  do {
    printf("> ");
    yyparse(&laMemoire,&valeurCalculee,&commande);
    if (commande==AFFICHER)
      printf("%lf\n",valeurCalculee);
  } while (commande!=QUITTER);
}
```

Références

- Bibliographie :

Alfred V. Aho, Ravi Sethi, Jeffrey Ullman, "Compilateurs, principes, techniques et outils", Addison Wesley 1986; ISBN 0-201-10088-6

Nino Silvero, "Réaliser un compilateur, les outils Lex et Yacc", Eyrolles, ISBN 2-212-08834-5

- Cours en ligne :

http://www.pps.jussieu.fr/~dicosmo/CourseNotes/Compilation/

http://www-adele.imag.fr/~donsez/cours/

http://fastnet.univ-brest.fr/~gire/COURS/COMPIL_IUP/POLY.html

Références

- Lex et Yacc :

  http://epaperpress.com/y_man.html

- ANTLR :

  http://www.antlr.org/