Compilation

Pr. Youness Tabii

Plan

- Introduction et rappel
- Analyseur lexical
- Analyseur syntaxique
- Analyseur sémantique
- Représentation intermédiaire
- Génération de code
- Optimisation

Introduction

- Les micro-processeurs deviennent indispensables et sont embarqués dans tous les appareils que nous utilisons dans la vie quotidienne
  - Transport
    - Véhicules, Systèmes de navigation par satellites (GPS), Avions, …
  - Télécom
    - Téléphones portables, Smart Phones, …
  - Électroménager
    - Machine à laver, Micro-ondes, Lave vaisselles, …
  - Loisir
    - e-book, PDA, Jeux vidéo, Récepteurs, Télévision, TNT, Home Cinéma…
  - Espace
    - Satellites, Navettes, Robots d'exploration, …
- Pour fonctionner, ces micro-processeurs nécessitent des programmes spécifiques à leur architecture matérielle

Introduction

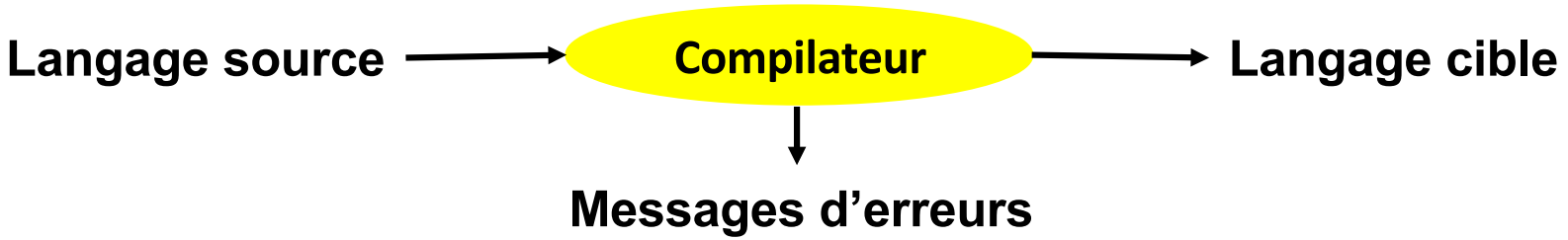
- Programmation en langages de bas niveau (proches de la machine)
  - **très difficile** (complexité)
    - Courbe d'apprentissage très lente
    - Débuggage fastidieux
  - **très coûteuse en temps** (perte de temps)
    - Tâches récurrentes
    - Tâches automatisables
  - **très coûteuse en ressource humaine** (budget énorme)
    - Tâches manuelles
    - Maintenance
  - **très ingrate** (artisanat)
    - Centrée sur les détails techniques et non sur les modèles conceptuels
  - **n'est pas à 100% bug-free** (fiabilité)
    - Le programmeur humain n'est pas parfait

Introduction

- Besoin continu de
  - Langages de haut niveau
    - avec des structures de données, de contrôles et des modèles conceptuels proches de l'humain
  - Logiciels de **génération** des **langages bas niveau** (propres aux microprocesseurs) à partir de **langages haut niveau** et vice versa
    - **Compilateurs**
    - **Interpréteurs**
    - **Pré-processeurs**
    - **Dé-compilateur**
    - **etc.**

Compilateur - Définition

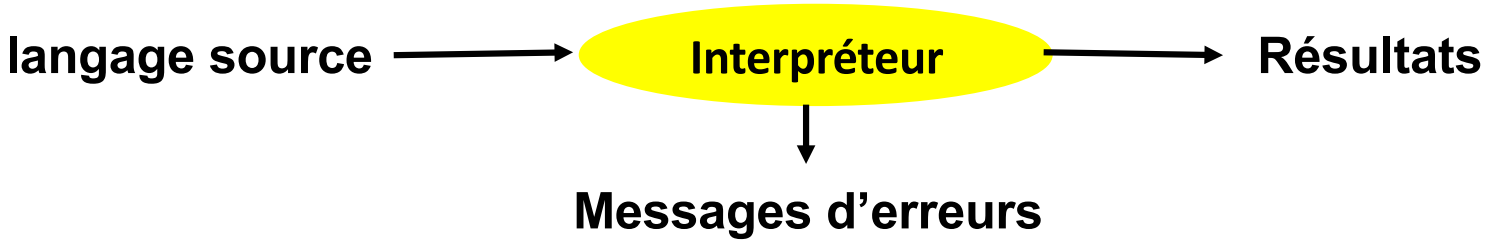
- Un compilateur est un logiciel (une *fonction*) qui prend en entrée un programme **P1** dans un langage source L1 et produit en sortie programme équivalent **P2** dans un langage L2



- Exemple de compilateurs :
  - C pour Motorola, Ada pour Intel, C++ pour Sun, doc vers pdf, ppt vers Postscript, pptx vers ppt, Latex vers Postscript, etc.

Interpréteur - Définition

- Un interpréteur est logiciel qui prend en entrée un programme **P1** dans un langage source L1 et produit en sortie les **Résultats** de l'exécution de ce programme



- Exemples d'interpréteurs :
  - Batch DOS, Shell Unix, Prolog, PL/SQL, Lisp/Scheme, Basic, Calculatrice programmable, etc.

Dé-compilateur

- Dé-Compilateur est un compilateur dans le sens inverse d'un compilateur (depuis le langage bas niveau, vers un langage haut niveau)
- Applications :
  - Récupération de vieux logiciels
    - Portage de programmes dans de nouveaux langages
    - Recompilation de programmes vers de nouvelles architectures
  - Autres
    - Compréhension du code d'un algorithme
    - Compréhension des clefs d'un algorithme de sécurité

Questions

- Quels sont les constituants d'un langage naturel ?

- Si l'on veut programmer un traducteur de l'Anglais vers le Français que proposeriez-vous comme algorithme ?

Quelques éléments de réponses

- Quelques constituants d'un langage naturel ?
  - Vocabulaire (lexique), Syntaxe (grammaire), Sémantique (sens selon le contexte)

- Un macro-algorithme pour traduire de l'Anglais vers le Français que proposeriez-vous comme ?
  1. Analyser les mots selon le dictionnaire Anglais
  2. Analyser la forme des phrases selon la grammaire de l'Anglais
  3. Analyser le sens des phrases selon le contexte des mots dans la phrase anglaise
  4. Traduire le sens des phrases dans la grammaire et le vocabulaire du Français

Étapes et Architecture d'un compilateur

1. Compréhension des entrées (**Analyse**)
   - Analyse des éléments lexicaux (lexèmes : mots) du programme : **analyseur lexical**
   - Analyse de la forme (structure) des instructions (phrases) du programme : **analyseur syntaxique**
   - Analyse du sens (cohérence) des instructions du programme : **analyseur sémantique**

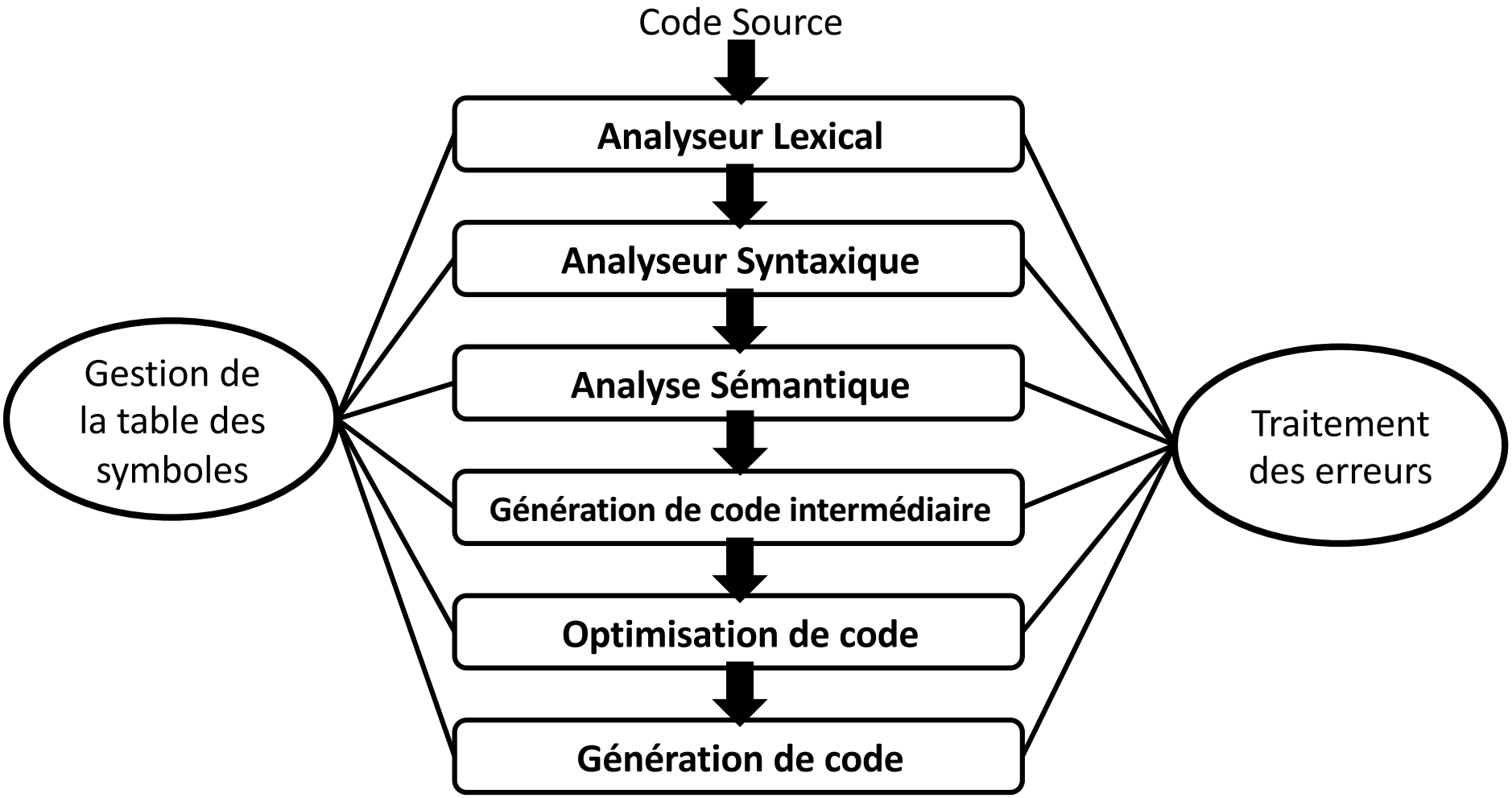
2. Préparation de la génération (**Synthèse**)
   - Génération d'un code intermédiaire (nécessitant des passes d'optimisation et de transcription en code machine) : **générateur de pseudo-code**
   - Optimisation du code intermédiaire : **optimisateur de code**

3. Génération finale de la sortie (**Synthèse suite**)
   - Génération du code cible à partir du code intermédiaire : **générateur de code**

Étapes et Architecture d'un compilateur

- ✓ Autres Fonctionnalités
  - ☐ gestion les erreurs et guide le programmeur pour corriger son programme : **gestionnaire d'erreurs**
  - ☐ gestion du dictionnaire des données du compilateur (symboles réservés, noms des variables, constantes) : **gestionnaire de la table des symboles**

Architecture d'un compilateur

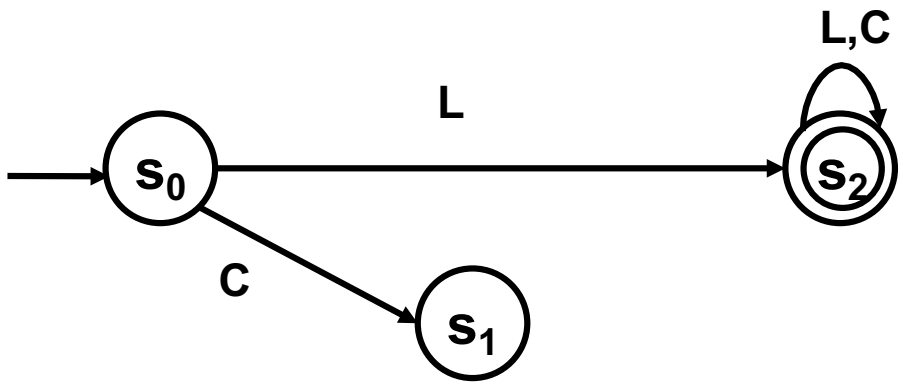


Rappel

Comment Transcrire une expression régulière en langage C ?

Expression : **L(L+C)*** Avec **L=[a-zA-Z]** et **C=[0-9]**

Rappel
ER → DFA minimal → Programme en C



Etat mort/Trash

Rappel
ER → DFA minimal → Programme en C

```
typedef  enum {S0, S1, S2} State;

int main(int argc,char *argv[]){
    // 1- traitement intial
    State state = S0;

    printf("S0\n");

    automate(state);

    return 0;
}
```

```
void automate(State current_state){
    State state = current_state;
    char c = getchar();
    if (c != '$') {
    // 2- traitement récursif
    switch (state){
            case S0 :
                if (((c >= 'a') && (c <='z')) || ((c >= 'A') && (c <='Z'))){
                    state = S1; printf("S0 --> S1\n");
                }else {state = S2; printf("S0 --> S2\n");}
                break;
            case S1 :
                if (((c >= 'a') && (c <='z')) || ((c >= 'A') && (c <='Z')) || ((c
 >= '0') &&  (c <= '9')))
                 state = S1; printf("S1 --> S1\n");
                break;
            case S2 :
                state = S2;  printf("S2 --> S2\n");
                }
    automate(state);
    }else{
   // 3- traitement final
   if (state == S1) printf("mot accepté par l'automate des identificateurs\n");
   else printf("mot refusé par l'automate des identificateurs\n");}}
```

Rappel
ER → DFA minimal → Programme en C

./idAutomate

state0 **a123456**$

state0 --> state1

state1 --> state1

state1 --> state1

state1 --> state1

state1 --> state1

state1 --> state1

state1 --> state1

mot accepté par l'automate des identificateurs

./idAutomate

state0 **12345**$

state0 --> state2

state2 --> state2

state2 --> state2

state2 --> state2

state2 --> state2

mot refusé par l'automate des identificateurs

Analyse Lexical

Définitions

- Le lexique d'un langage de programmation est son vocabulaire.
- Un lexème (token) est une collection de symboles élémentaires (un mot) ayant un sens pour le langage.
- Plusieurs lexèmes peuvent appartenir à une même classe. Une telle classe s'appelle une unité lexicale.
- L'ensemble des lexèmes d'une unité lexicale est décrit par une règle appelée modèle associé à l'unité lexicale.

Analyseur Lexical

- Le module qui effectue l'analyse lexicale s'appel un analyseur lexical (lexer ou scanner).
- Un analyseur lexical prend en entrée une séquence de caractères individuels et les regroupe en lexèmes.
- Autres tâches d'un analyseur lexical
  - Ignorer tous les éléments n'ayant pas un sens pour le code machine (Les espaces, les tabulations, les retours à la ligne, et les commentaires).

Définition

- **Unités lexicales (Tokens)**:
  - Symboles : identificateurs, chaînes, constantes numériques
  - Mots clefs : while, if, case, …
  - Opérateurs : <=, =, ==, +, -, …

Analyseur Lexical
Exemple

Input : a = b+c*2;

Ouput :

IDENTIFICATEUR
OPPAFFECT
IDENTIFICATEUR
PLUS
IDENTIFICATEUR
OPPMULT
CONSTANTE
PTVIRG

Tokens	Unité Lexical
a	Identificateur
=	Opérateur Affectation
b	Identificateur
+	Opérateur Plus
c	Identificateur
*	Opérateur Multiplication
2	Constante

Analyseur Lexical

✓ Les meilleurs modèles qui existent pour identifier les types lexicaux de tokens sont les expressions régulières

- **Alphabétique : (‘a’ | ‘b’ | … | ‘z’ | ‘A’ | ‘B’ | … | ‘Z’)**
- **Numérique : (0 | …| 9)**
- **Opérateurs : (+ | - | / | * | = | <= | >= | < | >)**
- **Naturel : Numérique+**
- **Entier : (+ | - | ε) Naturel**
- **Identificateur : (Alphabétique (Alphabétique | Numérique)*)**
- **ChaîneAlphaNumérique : " (Alphabétique | Numérique)+ "**

Analyseur Lexical

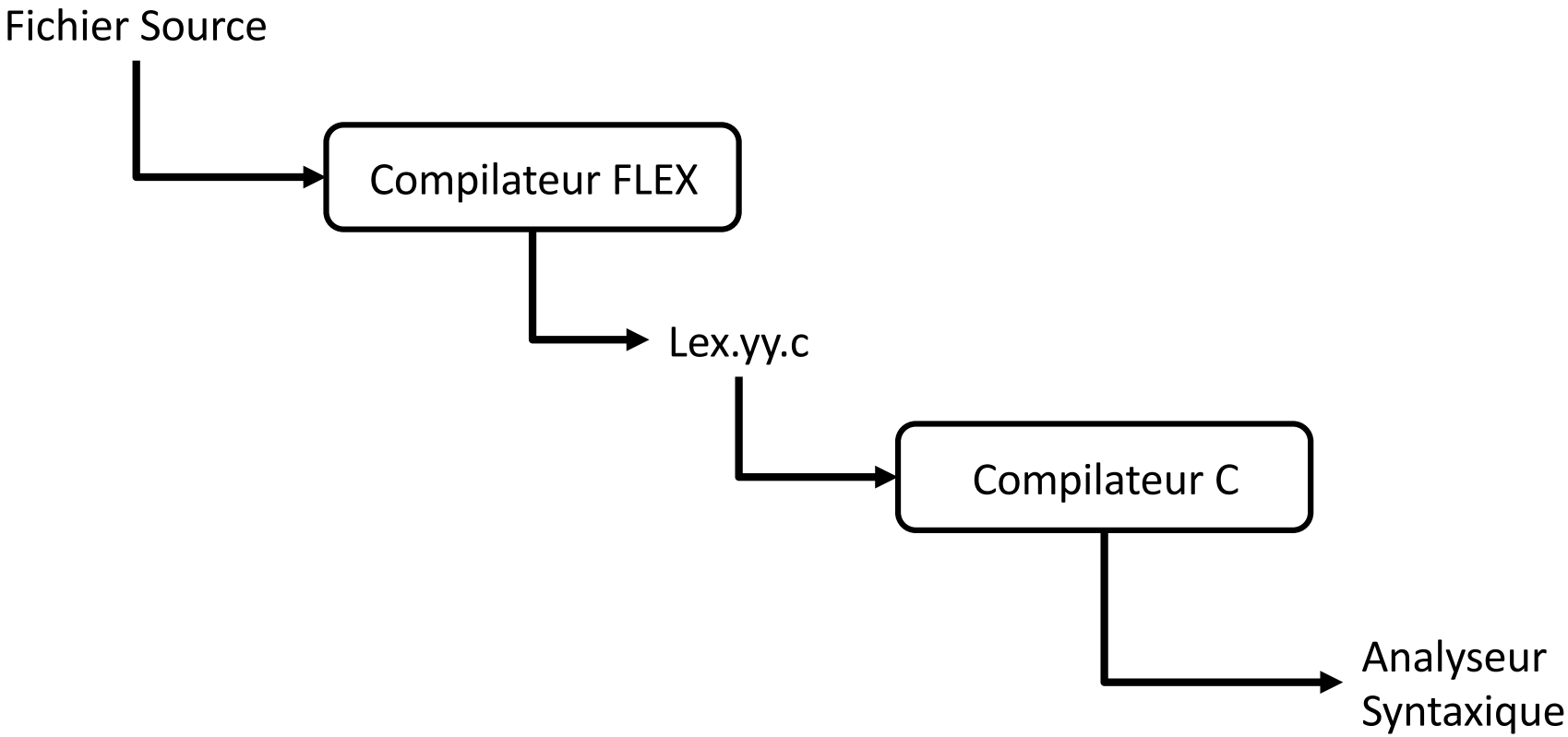
Il existe des algorithmes et outils permettant de générer le code implantant l'automate d'états finis (scanner) correspondant à une expression régulière (Lex)

Générateur de scanner - FLEX

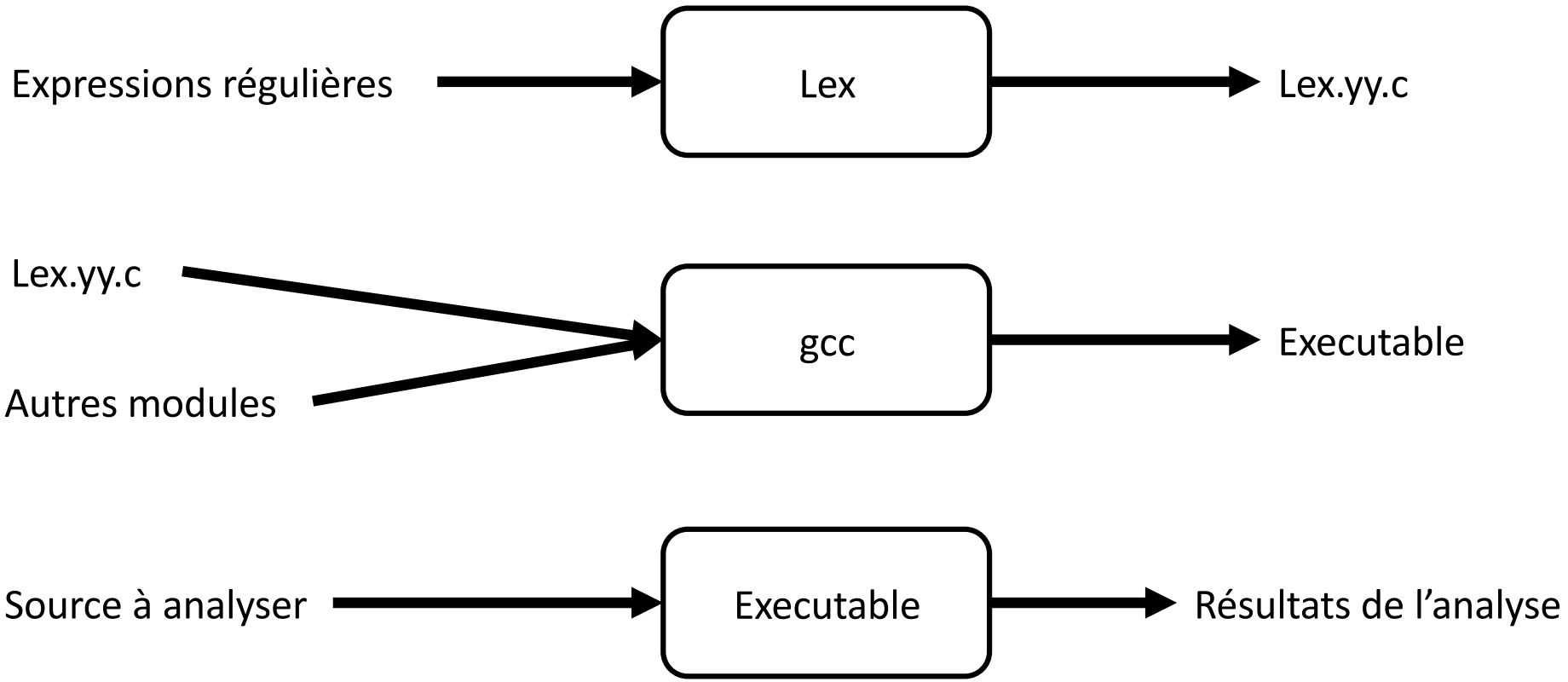
FLEX (FAST LEX)

- Flex est un outil de génération automatique d'analyseurs lexicaux.
- Un fichier Flex contient la description d'un analyseur lexical à générer.
  - Cette description est donnée sous la forme d'expressions régulières étendues et du code écrit en langage C (ou C + +).
- Flex génère comme résultat un fichier contenant le code C du future analyseur lexical (nommé lex.yy.c).
- La compilation de ce fichier par un compilateur C, génère finalement le code exécutable de l'analyseur lexical en question.

FLEX (FAST LEX)



FLEX
Etapes


Expressions régulières
Lex
Lex.yy.c
Lex.yy.c
Autres modules
gcc
Executable
Source à analyser
Executable
Résultats de l'analyse

FLEX

- Lorsque l'exécutable est mis en œuvre, il analyse le fichier source pour chercher les occurrences d'expressions régulières.
- Lorsqu'une régulière est trouvée, il exécute le code C correspondant.

Structure d'un programme FLEX

```
%{
/* Déclarations */
%}
/* Définitions */
%%
/* Règles */
%%
/* Code utilisateur */
```

- Déclaration/Code utilisateur: du C tout à fait classique
- Définitions : Expressions rationnelles auxquelles on attribue un nom
- Règles de production : Associations ER → code C à exécuter

Variables FLEX

- Dans les actions, on peut accéder à certaines variables spéciales :
  - **yylex()** : est la fonction principale du programme LEX.
  - **yyleng** : contient la taille du *token* reconnu ;
  - **yytext** : est une variable de type char* qui pointe vers la chaîne de caractères reconnue par l'expression régulière.
  - **yylval** : qui permet de passer des valeurs entières à YACC.
  - Il existe aussi une action spéciale : **ECHO** qui équivaut à **printf("%s",yytext).**
  - **yyin** : entrée du scanner
  - **yyout** : sortie du scanner

Expression régulière FLEX

Symbole	Signification
x	Le caractere 'x'
.	N'importe quel caractere sauf \n
[xyz]	Soit x, soit y, soit z
[^bz]	Tous les caracteres, SAUF b et z
[a-z]	N'importe quel caractere entre a et z
[^a-z]	Tous les caracteres, SAUF ceux compris entre a et z
R*	Zero R ou plus, ou R est n'importe quelle expression reguliere
R+	Un R ou plus
R?	Zero ou un R (c'est-a-dire un R optionnel)
R{2,5}	Entre deux et cinq R
R{2,}	Deux R ou plus
R{2}	Exactement deux R
"[xyz\"foo"	La chaine '[xyz"foo'
{NOTION}	L'expansion de la notion NOTION definie plus haut
\X	Si X est un 'a', 'b', 'f', 'n', 'r', 't', ou 'v', represente l'interpretation ANSI-C de \X.
\0	Caractere ASCII 0
\123	Caractere ASCII dont le numero est 123 EN OCTAL
\x2A	Caractere ASCII en hexadecimal
RS	R suivi de S
R|S	R ou S
R/S	R, seulement s'il est suivi par S
^R	R, mais seulement en debut de ligne
R$	R, mais seulement en fin de ligne
<<EOF>>	Fin de fichier

FLEX
Commandes

- Les étapes à suivre pour obtenir un analyseur avec Flex :

1. Ecrire votre analyseur dans un fichier portant une extension **".l"** ou **".lex"**. Par exemple, "**prog.lex**".
2. Compiler votre analyseur par la commende flex : **flex prog.lex**
3. Compiler le fichier par la commande gcc le fichier lex.yy.c produit par l'étape précédente :
   gcc lex.yy.c –lfl –o prog
4. Lancer l'analyseur en utilisant le nom de celui-ci : **prog**

Exemple

```
/*reconnaître les chiffres, lettres, réels et identificateurs*/
%{
/* Déclaration VIDE */
%}
%option noyywrap

/* Définition */
chiffre [0-9]
lettre  [a-zA-Z]
entier {chiffre}+
reel {chiffre}+(\.{chiffre}+(e(\+|\-)?{chiffre}+)?)?
ident {lettre}({lettre}|{chiffre})*

%%
/* Régles exrp → action*/
{entier} printf("\n entier %s \n ", yytext);
{reel}   printf("\n reel %s \n ", yytext);
{ident}  printf("\n identificateur %s \n ", yytext);
%%

int main(void){
        yylex();
        return 0;}
```

12 → entier 12

3.4→ reel 3.4

L → identificateur L

6.5e+4 → reel 6.5e+4

Exercice 1

- Compte le nombre de Voyelles, Consonnes et les caractères de ponctuation d'un texte entré en clavier.

- Etapes :
  - Déclaration des compteurs
  - Définition des expressions régulières (les consonnes, voyelles et ponctuation)
  - L'action à exécuté pour chaque expression régulière
  - La fonction main (pour affichage des résultat)

Solution

```
%{
int nbConsonnes,nbVoyelles,nbPonctuation;
%}

%option noyywrap
consonne [b-df-hj-np-xz]
ponctuation    [,;:?!\.]

%%

[aeiouy] nbVoyelles++;
{consonne} nbConsonnes++;
{ponctuation} nbPonctuation++;
.|\n     // ne rien faire

%%

int main(void){
        nbConsonnes=0;
        nbVoyelles=0;
        nbPonctuation=0;
        yylex();
        printf("\n Nb Consonnes : %d,
                    Nb Voyelles : %d,
                    Nb Poncuation : %d \n",
                    nbConsonnes,nbVoyelles,nbPonctuation);
       return 0;
        }
```

$ bonjour,

Nb Consonnes : 4, Nb Voyelles : 3, Nb Ponctuation : 1

Fin Séance 1

Analyse Syntaxique

Analyseur Syntaxique -Parser-

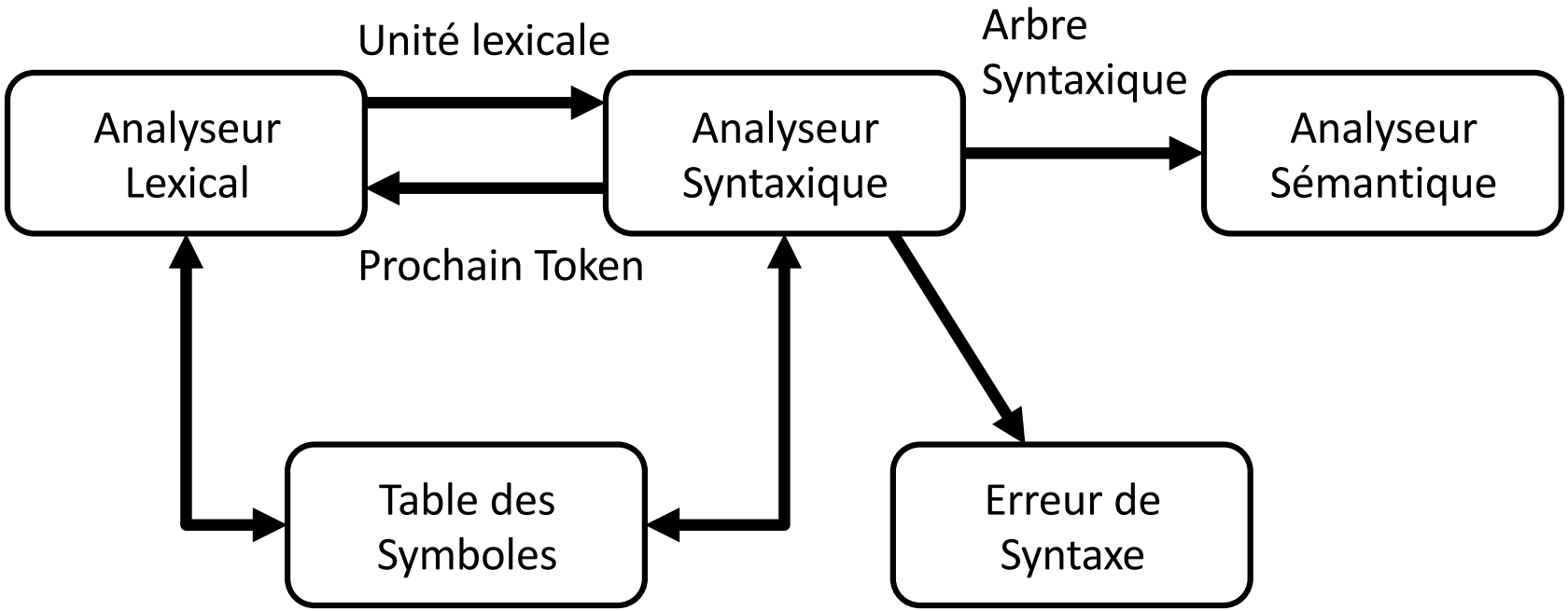
- L'analyseur syntaxique vérifie que l'ordre des tokens correspond à l'ordre définit pour le langage. On dit que l'on vérifie la syntaxe du langage à partir de la définition de sa grammaire.

- L'analyse syntaxique produit une représentation sous forme d'arbre de la suite des tokens obtenus lors de l'analyse lexicale

Analyseur Syntaxique

- Pour effectuer efficacement une analyse syntaxique, le compilateur nécessite :
  - Une définition formelle du langage source,
  - Une fonction indicatrice de l'appartenance d'un programme au langage source,
  - Un plan de gestion des entrées illégales.

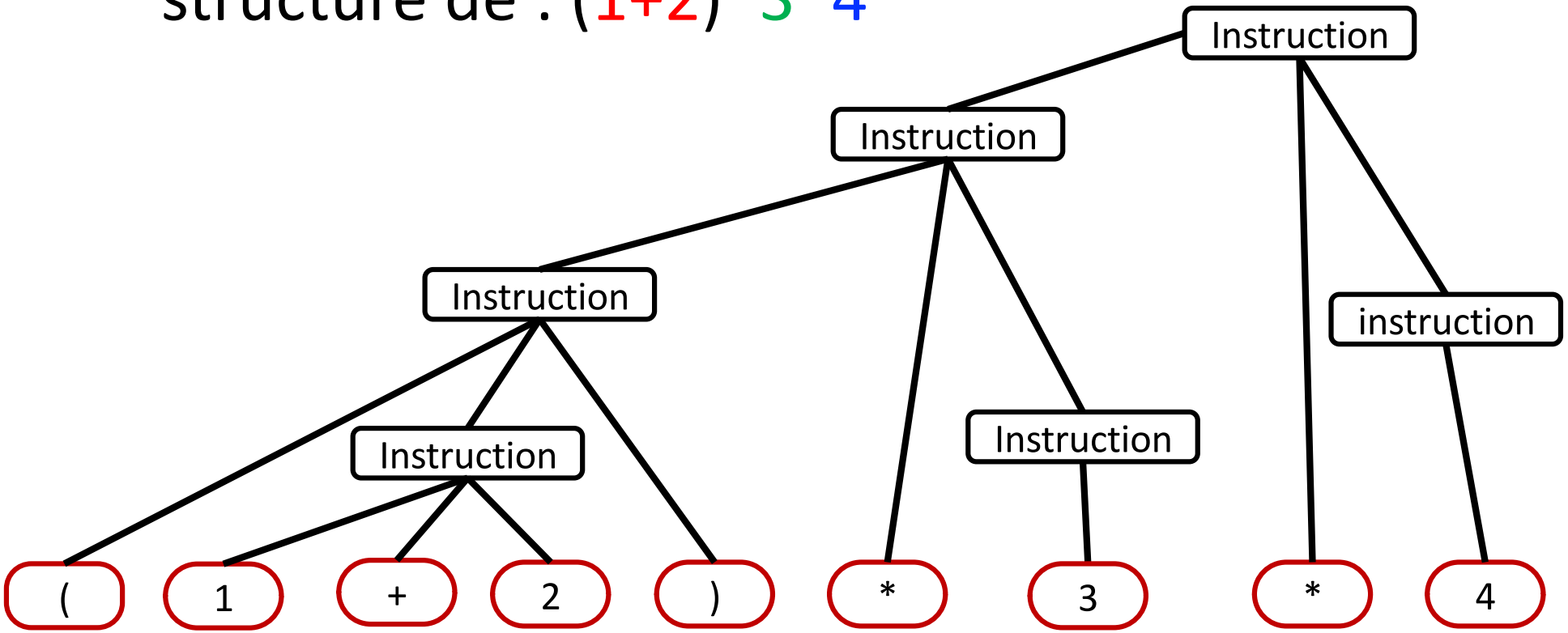
Analyseur Syntaxique



Analyseur Syntaxique

Arbre syntaxique - Exemple

- Arbre syntaxique suivant représente la structure de : (1+2)*3*4



Syntaxe et Grammaire

- La syntaxe est traditionnellement exprimée à l'aide d'une ***grammaire***

- Une ***grammaire*** **G** est une ***collection de règles de réécriture*** qui définissent mathématiquement quand une suite de symbole d'un certain alphabet constitue un mot d'un langage

- *L'ensemble des mots* pouvant être dérivées de ***G*** est appelé le *langage défini par* ***G***, noté ***L(G).***

Grammaire

- Une **grammaire** est formellement définie par :
  - Un ensemble de symboles **terminaux** (token) : Les symboles élémentaires du langage.
  - Un ensemble de symboles **non-terminaux**
  - Un ensemble de **règles syntaxiques** (ou de productions). Tête → terminaux et/ou non-terminaux
  - Un **axiome** (symbole initial, un non-terminal).
- Une **grammaire** définit un langage formé par l'ensemble des séquences finies de symboles **terminaux** qui peuvent être dérivées de l'**axiome** par des applications successives des productions.

Grammaire
Exemple

- **Exemple d'une structure du if en langage C :**

- <structure_if> ::= if «(» <condition> «)» «{» <instruction> «}»

  - **<structure_if>, <condition>**, **<instruction>** : **non-terminaux**.
  - **::=** : est un **méta-symbole** (symbole de la grammaire) signifiant «est défini par».
  - **if**, **«(»** , **«)»** , **«{»** et **«}»** : **des terminaux**. Lorsque les **terminaux** ne font qu'un caractère, ou qu'ils contiennent des caractères non alphanumériques, ou qu'ils peuvent être confondus avec des méta-symboles, ils sont écrits entre guillemets.

Grammaire
Exemple

```
G1 = (  {« a », « b », « c », « d »,        /* terminaux */
        « + », « - », « * », « / »,
        « ( » , « ) », « ^ »},
        {<expression>, <facteur>  },/* non-terminaux */
        {<expression> ::= <facteur>,          /* productions */
         <expression> ::= <expression> « + » <expression>,
         <expression> ::= <expression> « - » <expression>,
         <expression> ::= <expression> « * » <expression>,
         <expression> ::= <expression> « / » <expression>,
         <facteur> ::= « a »,
         <facteur> ::= « b »,
         <facteur> ::= « c »,
         <facteur> ::= « d »,
         <facteur> ::= « ( » expression « ) »,
         <facteur> ::= <facteur> « ^ » <facteur> },
      <expression>                                  /* axiome */)
```

Grammaire

Résumé

- Une grammaire dérive des chaînes en commençant par l'axiome et en remplaçant de façon répétée les non-terminaux décrits par les productions de la grammaire.
- Les chaînes de terminaux dérivables à partir de l'axiome forment le langage défini par la grammaire.

Grammaire hors-Contexte (GHC)

- Une Grammaire hors-contexte (GHC) ***G*** *est un 4-uplet* ***G = <T, NT, S, P>*** *où* :
  - ***T*** est l'ensemble des symboles *terminaux* (concrets) ou lettres de l'alphabet
  - ***NT*** est l'ensemble des symboles non-*terminaux* (abstraits)
  - ***S*** $\in$ ***NT****, appelé symbole initial (Start ou axiome)*
    - Toute dérivation d'un mot de **L(G)** débute par **S**
    - À partir de ***S***, on dérive l'ensemble des mots de ***L(G)***
  - ***P*** est l'ensemble des règles de réécriture ou de production. Formellement, une règle de ***P*** est sous la forme : ***NT*** $\rightarrow$ ***(T*** $\cup$ ***NT)****

Langage dérivé

- Soit ***G = <T, NT, S, P>*** une grammaire. On appelle **langage engendré** par ***G*** l'ensemble $L(G) = \{w \in T^* / S \Rightarrow_{r \in P}^{+} w\}$
  - *où* $\Rightarrow_{r \in P}$, *est appelée dérivation et dénote l'application d'une règle de production* ***r*** *de* ***P***
  - ***et*** $\Rightarrow_{r \in P}^{+}$ *dénote la répétition de règles* $\Rightarrow_{r \in P}$

Grammaire hors-Contexte (GHC)
Exemple

- *Soit* G=<T,NT,S,P> *avec:*
  - ✓T = {a,b}
  - ✓NT={S,A,B}
  - ✓S *: l'axiome.*
  - ✓P ={**S→AB|aS|A**, **A→Ab|ε**, **B→AS**}

- *Pour cette grammaire, les mots* **AB**, **aaS** *et* **ε** *sont des formes sur* G.

Grammaire hors-Contexte (GHC)
Exemple

- Une grammaire hors-contexte qui engendre les palindromes sur {*a*, *b*} :

- $S \rightarrow aSa$
- $\mid bSb$
- $\mid \varepsilon$

Grammaire hors-Contexte (GHC)
Ecriture

- G = <T={0,1}, NT={S}, S, P={r1,..,r5} >

- **1ère écriture des règles**
  - r1: S → ε
  - r2: S → 0
  - r3: S → 1
  - r4: S → 0 S 0
  - r5: S → 1 S 1

- **2ème écriture des règles**
  - r1: S → ε
  - r2: | 0
  - r3: | 1
  - r4: | 0 S 0
  - r5: | 1 S 1

- **3ème écriture des règles (BNF -Backus-Naur Form)**
  - r1: **<S> ::=** (ε n'est pas représentable : c'à dire : un vide = ε)
  - r2: | 0
  - r3: | 1
  - r4: | 0 **<S>** 0
  - r5: | 1 **<S>** 1

BNF-Backus-Naur Form
Exemple

Grammaire BNF pour la construction d'un langage naturel simple

```
<Phrase> ::= <sujet> <verbe> <complément>
<sujet> ::= <article> <adjectif> <nom> |
            <article> <nom> <adjectif> |
            <article> <nom>
<article>  ::= «le» | «la» | «l'» |
               «les» | «un» | «une» | «des»
<adjectif> ::= «grand» | «petit» | <couleur>
<couleur> ::= «bleu» | «vert» | «rouge»
<verbe> ::= (etc)
```

Grammaire hors-Contexte (GHC)
Exercice 1

- Quel est le langage ***L(G)*** décrit par la grammaire hors-contexte suivante ?

- ***G = < T ={a}, NT ={S}, S, P = {r1, r2} >***
  - **r1: S → aS**
  - **r2: | a**

Grammaire hors-Contexte (GHC)
Solution 1

- **G:**
  - **r1:** $S \rightarrow aS$
  - **r2:** $\mid a$

- Raisonnons par induction sur la taille des mots w de L(G)
- $|w| = 1 : S \Rightarrow_{r2} a$
- $|w| = 2 : S \Rightarrow_{r1} aS \Rightarrow_{r2} aa = a^2$
- $|w| = 3 : S \Rightarrow_{r1} aS \Rightarrow_{r1} aaS \Rightarrow_{r2} aaa = a^3$
- $|w| = 4 : S \Rightarrow_{r1} aS \Rightarrow_{r1} aaS \Rightarrow_{r1} aaaS \Rightarrow_{r2} aaaa = a^4$
- $|w| = i : S \Rightarrow_{r1} aS \Rightarrow_{r1} aaS \Rightarrow_{r1} aaS \Rightarrow_{r1} \ldots \Rightarrow_{r1} aaa..aaS \Rightarrow_{r2} aaa..aaa = a^i$

- $L(G) = \cup_{1 \leq i} \{w \in T^* / w = a^i\} = \{ w \in T^* / w = a+\}$
- ***L(G)*** est le langage de mots formés d'une suite non vide de la lettre 'a'

Grammaire hors-Contexte (GHC)
Exercice 2

On considère la grammaire G = <T,NT,S,P> où

$$T = \{b,c\}$$
$$NT = \{S\}$$
$$P = \{ S \rightarrow bS \mid cc\}$$

Déterminer L(G).

Grammaire hors-Contexte (GHC)
Solution 2

En effet, partant de l'axiome S, toute dérivation commencera nécessairement par appliquer 0, 1 ou plusieurs fois la première règle puis se terminera en appliquant la deuxième règle.

On représentera cela en écrivant le schéma de dérivation suivant :

$$S \Longrightarrow_{r1}^{*} b^n S \Longrightarrow_{r2} b^n cc \quad n \in \mathbb{N}$$

Alors L(G) = $\{b^n cc \ / \ n \in \mathbb{N}\}$

Grammaire hors-Contexte (GHC)
Exercice 3

On considère la grammaire G = <T,NT,S,P> où

T = { a,b,0}

NT = { S,U}

P = { S→ aSa | bSb | U

U → 0U | ε }

Déterminer L(G).

Grammaire hors-Contexte (GHC)
Solution 3

Prenons les cas suivants

$$S \Rightarrow_{r1} aSa \Rightarrow_{r1} aUa \Rightarrow_{r2}^{n} a0^n a$$

$$S \Rightarrow_{r1} aSa \Rightarrow_{r1} abSba \Rightarrow_{r2}^{n} ab0^n ba$$

ab0^{n}ba

On définit inverse(u) est le mot inverse de u, tel que v=inverse(u)

Alors L(G) = $\{u0^n v / u \in \{a,b\}^*, v = inverse(u), n \in \mathbb{N}\}$

Grammaire hors-Contexte (GHC)
Exercice 4

On considère la grammaire G = <T,NT,Ph,P> où

T = { un , une , le , la , enfant , garçon , fille , cerise , haricot , cueille , mange }
NT = {Ph,Gn,Gv,Df,Dm,Nf,Nm,V }
P = {
Ph→Gn Gv
Gn→Df Nf | Dm Nm
Gv → V Gn
Df → une | la
Dm→ un | le
Nf→ fille | cerise
Nm → enfant | garçon | haricot
V → cueille | mange

}

- La phrase "**une cerise cueille un enfant**" appartient-elle au langage L(G)?

Grammaire hors-Contexte (GHC)
Solution 4

Pour montrer qu'une phrase appartient au langage, on construit une dérivation de l'axiome Ph jusqu'à la phrase.

On souligne à chaque fois le symbole non terminal qui est remplacé par la dérivation.

Ph ⇒ Gn Gv ⇒ Df Nf Gv ⇒ Df Nf V Gn

⇒ Df Nf V Dm Nm ⇒ une Nf V Dm Nm

⇒ une cerise V Dm Nm

⇒ une cerise cueille Dm Nm

⇒ une cerise cueille un Nm

⇒ une cerise cueille un enfant

P = {Ph→Gn Gv
Gn→Df Nf | Dm Nm
Gv → V Gn
Df → une | la
Dm→ un | le
Nf→ fille | cerise
Nm → enfant | garçon | haricot
V → cueille | mange }

GHC Linéaire Droite

- Une grammaire ***G = <T, NT, S, P>*** *HC est dite* :

  - ***Linéaire Droite :*** si l'ensemble de ses règles de réécriture ***P*** *sont de la* forme : ***NT → (T ∪ T.NT)***
    - La partie droite des règles de récriture contient un symbole terminal OU un symbole terminal suivi d'un symbole non-terminal
    - e.g. **G : S → aS | a**

GHC Linéaire Gauche

- Une grammaire ***G = <T, NT, S, P>*** *HC est dite* :
  - ***Linéaire Gauche :*** si l'ensemble de ses règles de réécriture ***P*** *sont de la* forme : ***NT → (T ∪ NT.T)***
    - La partie droite des règles de récriture contient un symbole terminal OU un symbole non-terminal suivi d'un symbole terminal
    - e.g. **G : S → Sa | a**

Langages réguliers et Grammaire

- **Théorèmes** :
  - Toute **grammaire HC Linéaire Droite G** génère un **langage régulier L(G)** (L(G) est reconnu par un automate d'état fini)
    - e.g. **G : S → aS | a**
  - Tout **langage régulier L** possède une **grammaire HC Linéaire Droite G** (L(G) = L)

Algorithme DFA → GHC

Principe de la construction :

- *Soit le DFA* $M = \langle Q, T, \delta, q_0, F \rangle$
  - *si* $q_0 \notin F$ *: on définit* $G = \langle T, Q, q_0, P \rangle$ *équivalent avec*
    - ✓ $p \to aq \in P \iff \delta(p,a)=q$
    - ✓ $p \to a \in P \iff \delta(p,a) \in F$
  - *si* $q_0 \in F$ *: on fait comme le cas précédant + on rajoute la variable S et*
    - ✓ $S \to q_0 \mid \varepsilon$

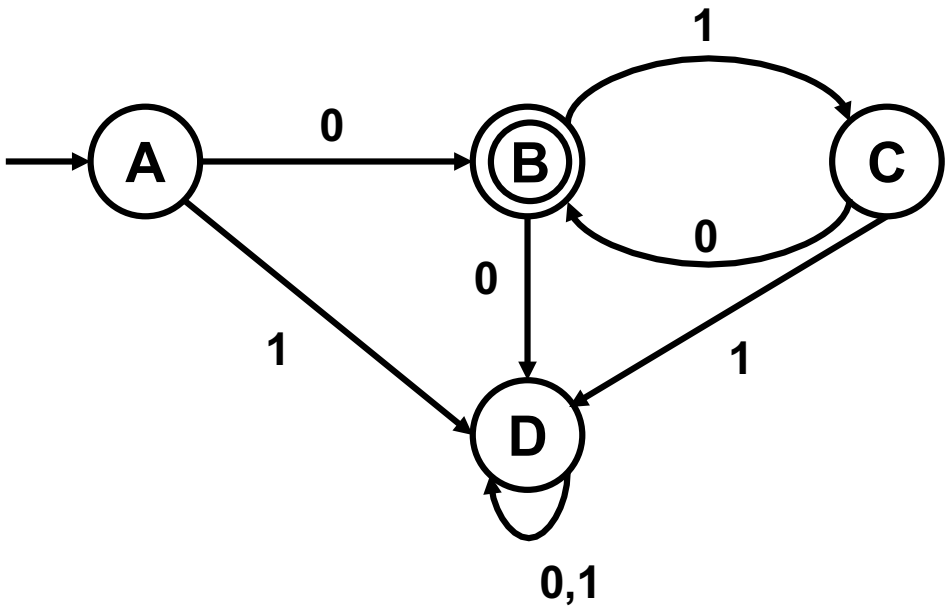
Algorithme DFA → GHC
Autrement dit

Input : A = <S, Σ, δ, s_0, F >
Output : G = <T, NT, S, P>

- On fait correspondre
  - A chaque **s** de **S**, un élément de **NT** (NT(s))
  - A chaque élément **l** de Σ, un élément de **T** (T(l))
  - A chaque élément **(s,l,s')** de δ, un élément de P (P(s,l,s'))
  - A s_0 le non terminal **S**
  - A chaque élément **s** de **F**, une règle **NT**(**s**)--> **ε**

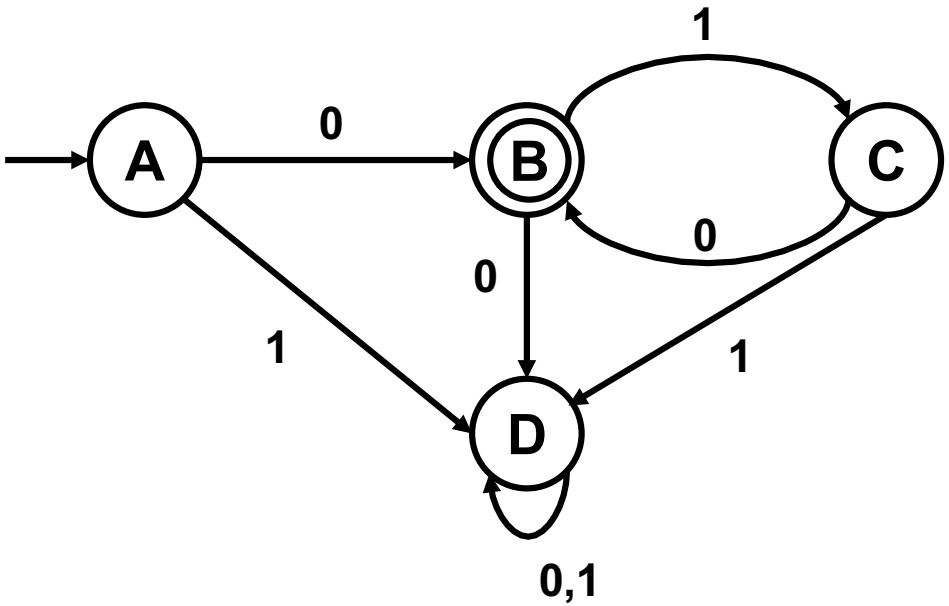
Algorithme DFA → GHC
Exercice

Trouvez la grammaire linéaire droite correspondante



Algorithme DFA → GHC
Solution

A→ 0B |1D |0
B → 0D |1C
C → 0B|1D|0
D → 0D |1D



Après élimination de *D* inutile :

A→ 0B|0
B → 1C
C → 0B|0

Fin Séance 2

Dérivation : Gauche & Droite
Exemple

- Soit la grammaire HC **G3 = <**
  - ✓ ***T={Number,+,-, ×, ÷ },***
  - ✓ ***NT ={Expr, Op},***
  - ✓ ***S=Expr,***
  - ✓ ***P = {r1,..,r6} >***
    - ✓ r1 : Expr → Expr Op Number
    - ✓ r2: | Number
    - ✓ r3 : Op → +
    - ✓ r4 : |-
    - ✓ r5 : |×
    - ✓ r6 : | ÷

Dérivation : Gauche & Droite
Exemple

Exemples de *dérivations gauches* de mots dans ***L(G3)***

1: $\boldsymbol{Expr} \Rightarrow_{r1} \boldsymbol{Expr}\ Op\ Number \Rightarrow_{r2} Number\ \boldsymbol{Op}\ Number \Rightarrow_{r3} Number + Number$

2: $\boldsymbol{Expr} \Rightarrow_{r1} \boldsymbol{Expr}\ Op\ Number \Rightarrow_{r1} \boldsymbol{Expr}\ Op\ Number\ Op\ Number$

$\Rightarrow_{r2} Number\ \boldsymbol{Op}\ Number\ Op\ Number \Rightarrow_{r3} Number + Number\ \boldsymbol{Op}\ Number$

$\Rightarrow_{r5} Number + Number * Number$

Dérivation : Gauche & Droite
Exercice

Dérivation à droite de l'expression Number+Number*Number

Dérivation à Droite

$$\boldsymbol{Expr} \Rightarrow_{r1} Expr\ \boldsymbol{Op}\ Number \Rightarrow_{r5} \boldsymbol{Expr} * Number \Rightarrow_{r1} Expr\ \boldsymbol{Op}\ Number * Number$$

$$\Rightarrow_{r3} \boldsymbol{Expr} + Number * Number \Rightarrow_{r2} Number + Number * Number$$

Règles appliquées : (r1, r5, r1, r3, r2)

Mêmes règles mais pas dans le même ordre

Dérivation à Gauche

$$\boldsymbol{Expr} \Rightarrow_{r1} \boldsymbol{Expr}\ Op\ Number \Rightarrow_{r1} \boldsymbol{Expr}\ Op\ Number\ Op\ Number \Rightarrow_{r2} Number\ \boldsymbol{Op}\ Number\ Op\ Number$$

$$\Rightarrow_{r3} Number + Number\ \boldsymbol{Op}\ Number \Rightarrow_{r5} Number + Number * Number$$

Règles appliquées : (r1, r1, r2, r3, r5)

Analyseur Syntaxique

- L'analyseur syntaxique doit découvrir automatiquement pour une expression donnée en langage ***L***, son arbre syntaxique de dérivation par rapport à la grammaire de ce langage ***L***

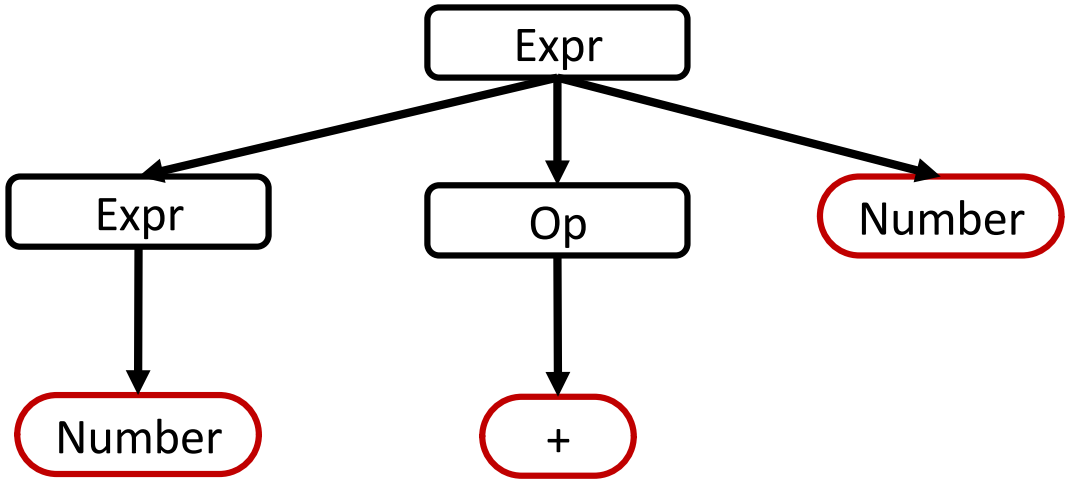
  - La **racine de l'arbre** syntaxique est : le symbole non-terminal initial **S**

  - Les **nœuds de l'arbre** syntaxique sont : le résultat de l'application des règles de production $\in (T \cup NT)*$

  - Les **feuilles de l'arbre** syntaxique sont : les éléments de l'expression donnée en entrée $\in T*$

Arbre Syntaxique (Arbre de dérivation)

$$\boldsymbol{Expr} \Rightarrow_{r1} \boldsymbol{Expr}\ Op\ Number \Rightarrow_{r2} Number\ \boldsymbol{Op}\ Number \Rightarrow_{r3} Number + Number$$

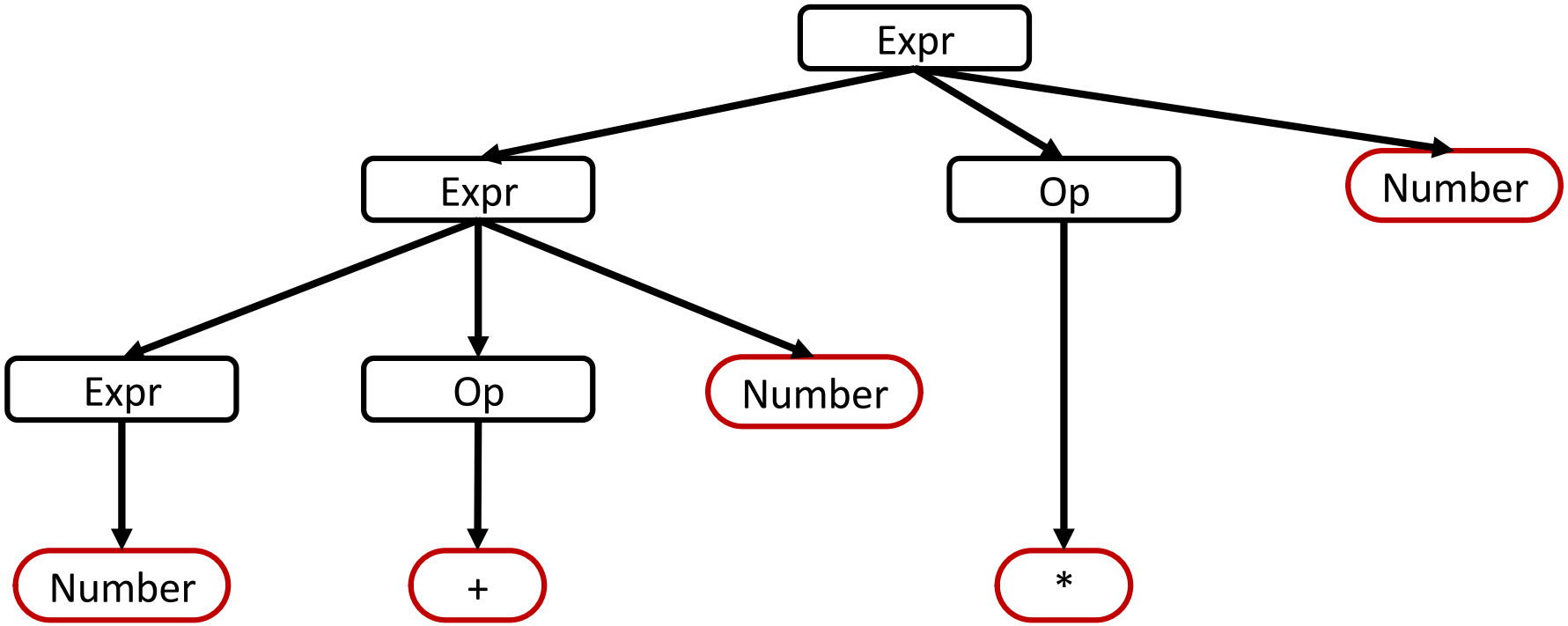


Arbre Syntaxique (Arbre de dérivation)

Dérivation à Gauche

$$\boldsymbol{Expr} \Rightarrow_{r1} \boldsymbol{Expr}\ Op\ Number \Rightarrow_{r1} \boldsymbol{Expr}\ Op\ Number\ Op\ Number \Rightarrow_{r2} Number\ \boldsymbol{Op}\ Number\ Op\ Number$$

$$\Rightarrow_{r3} Number + Number\ \boldsymbol{Op}\ Number \Rightarrow_{r5} Number + Number * Number$$

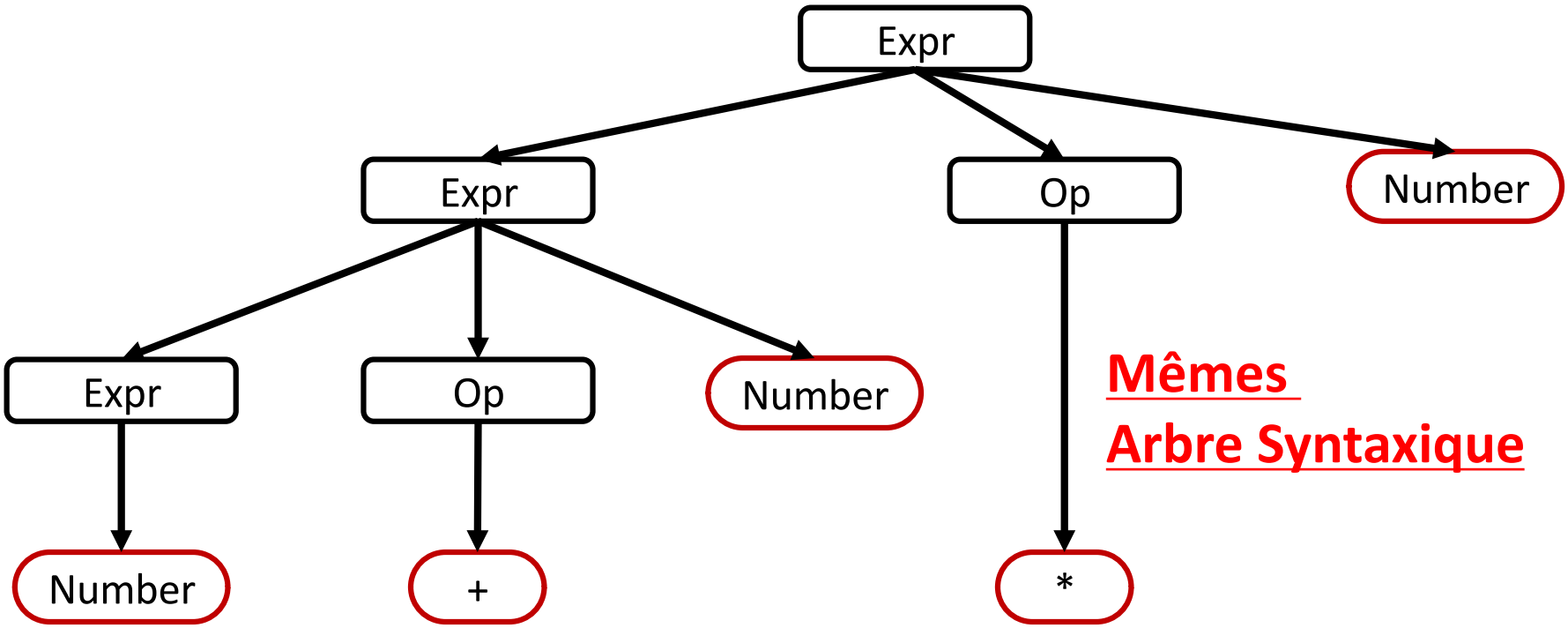


Arbre Syntaxique (Arbre de dérivation)

Dérivation à Droite

$$\boldsymbol{Expr} \Rightarrow_{r1} Expr\ \boldsymbol{Op}\ Number \Rightarrow_{r5} \boldsymbol{Expr}\ *\ Number \Rightarrow_{r1} Expr\ \boldsymbol{Op}\ Number\ *\ Number$$

$$\Rightarrow_{r3} \boldsymbol{Expr} + Number\ *\ Number \Rightarrow_{r2} Number + Number\ *\ Number$$



Grammaire hors-contexte Ambigüe

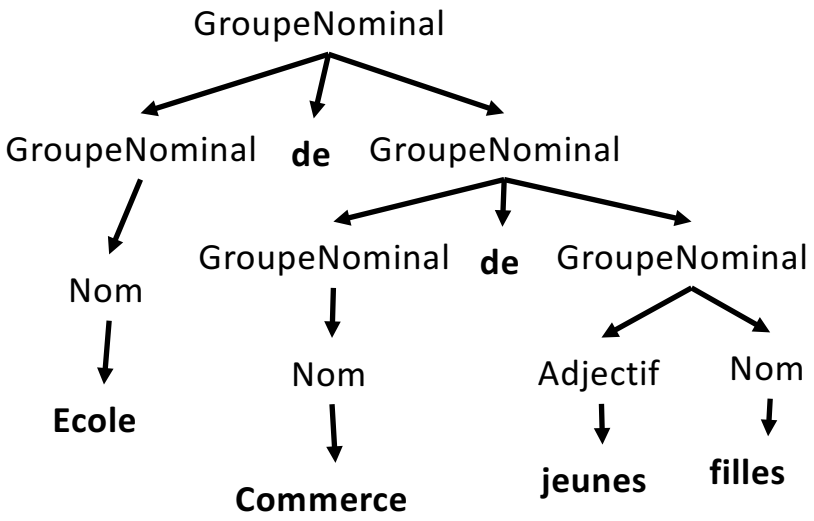
Grammaire ambigüe

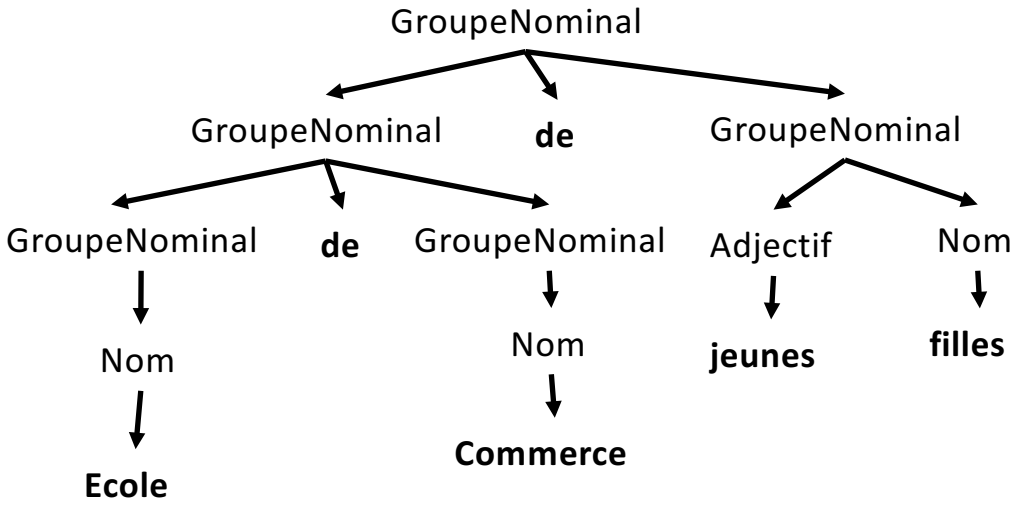
- Pour une CFG *G* tout string *w* de *L*(*G*) a au moins un arbre de dérivation pour *G*.
- $w \in L(G)$ peut avoir plusieurs arbres de dérivation pour *G* : dans ce cas on dira que la grammaire est ambigüe.
- Idéalement, pour permettre le parsing, une grammaire ne doit pas être ambigüe. En effet, l'arbre de dérivation détermine le code généré par le compilateur.

Grammaire ambigüe
Exemple

- Soit les règles de la grammaire suivante :
  - Grammaire Simpliste :
    - r1: **GroupeNominal** → Nom
    - r2 : | **Adjectif** Nom
    - r3 : | **GroupeNominal** de **GroupeNominal**
  - Trouver un arbre de dérivation du mot
    - w = « **Ecole de Commerce de jeunes filles** »

Grammaire ambigüe
Exemple

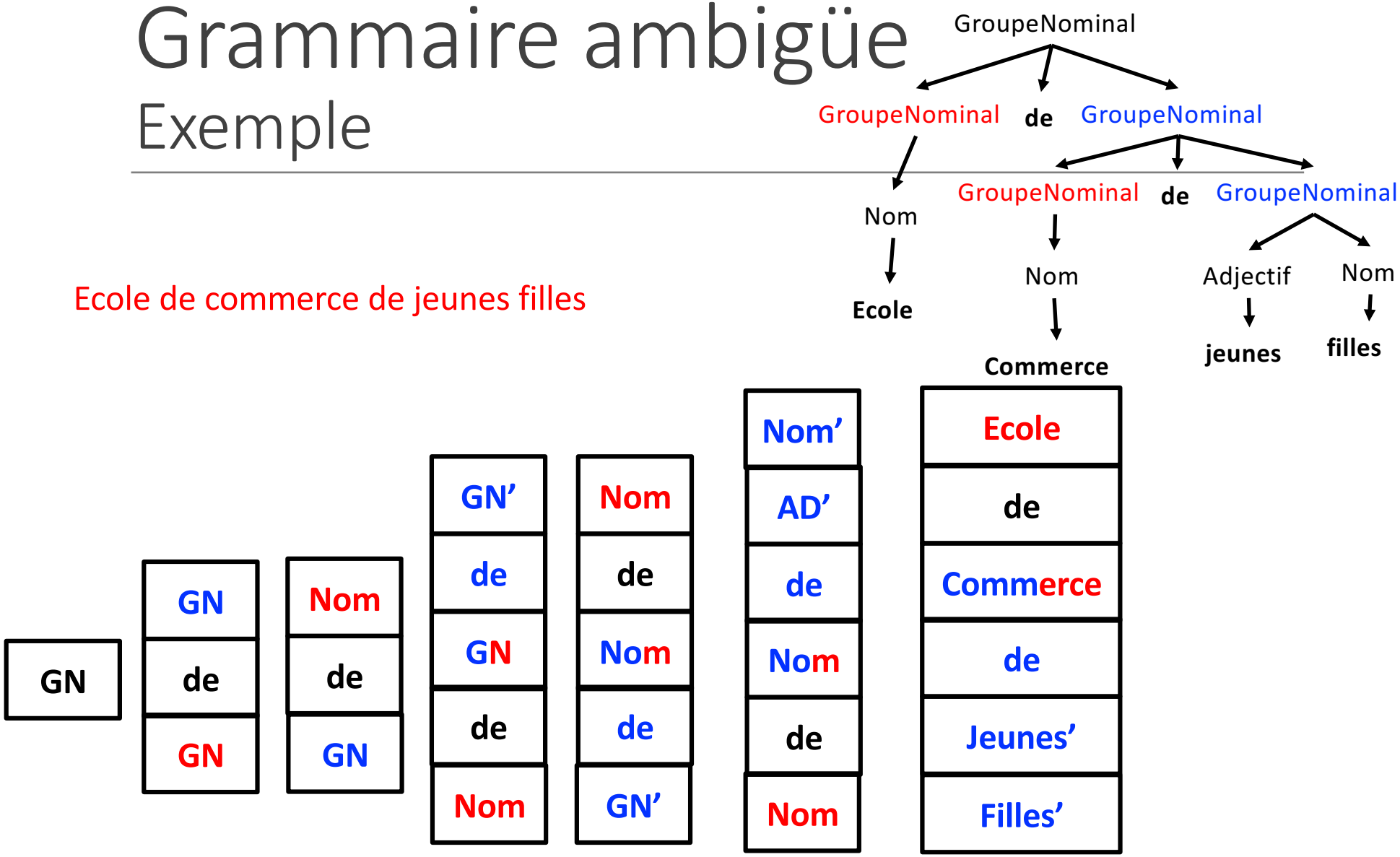




2 arbres syntaxiques différents d'une expression (i.e. **ambigüité de la grammaire**) ⇒ **2** interprétations différentes ⇒ **2** sémantiques différentes ⇒ **2** codes possibles à générer pour cette expression ⇒ **Problème de non-déterminisme pour le compilateur !!**

Grammaire ambigüe

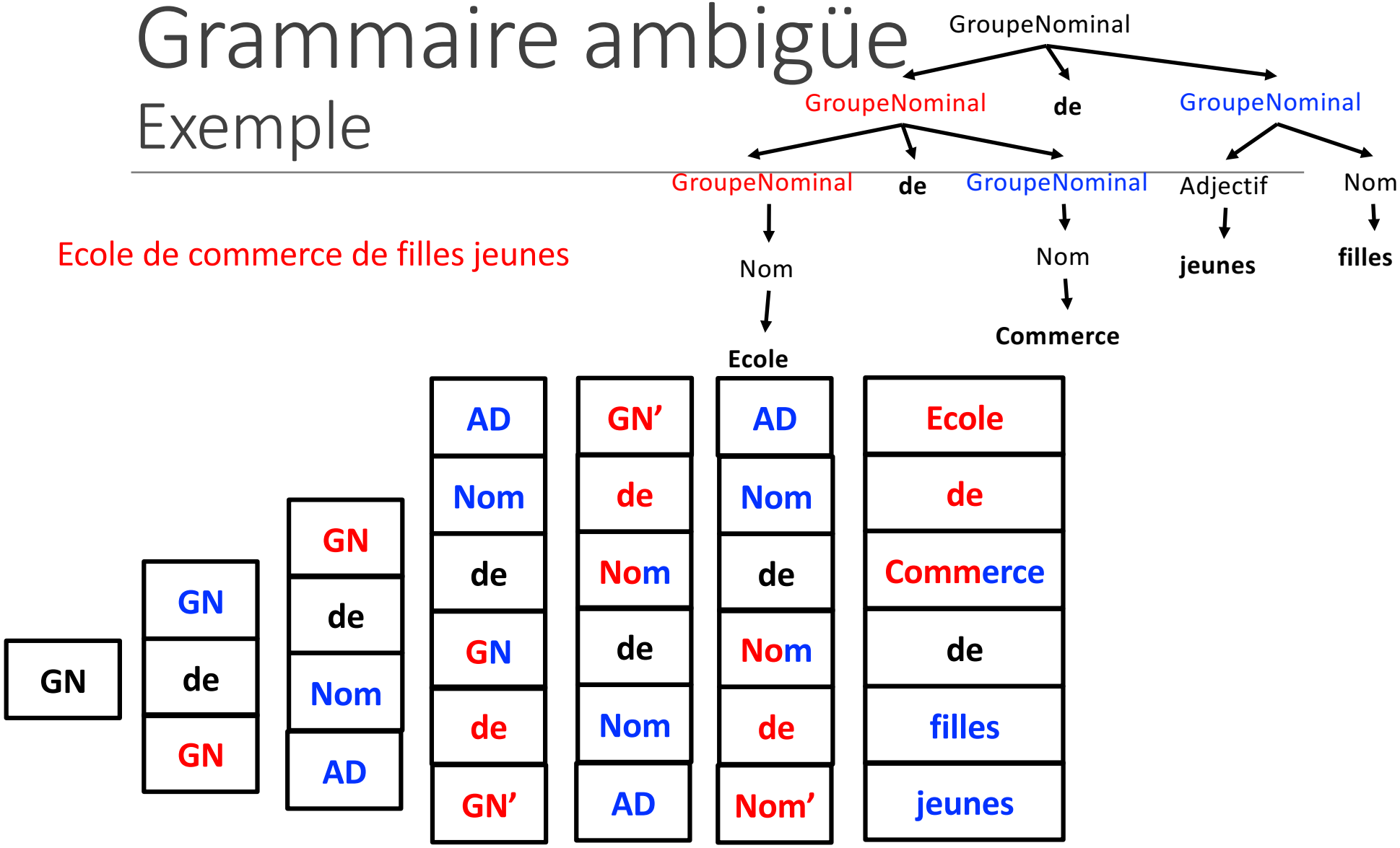
Exemple



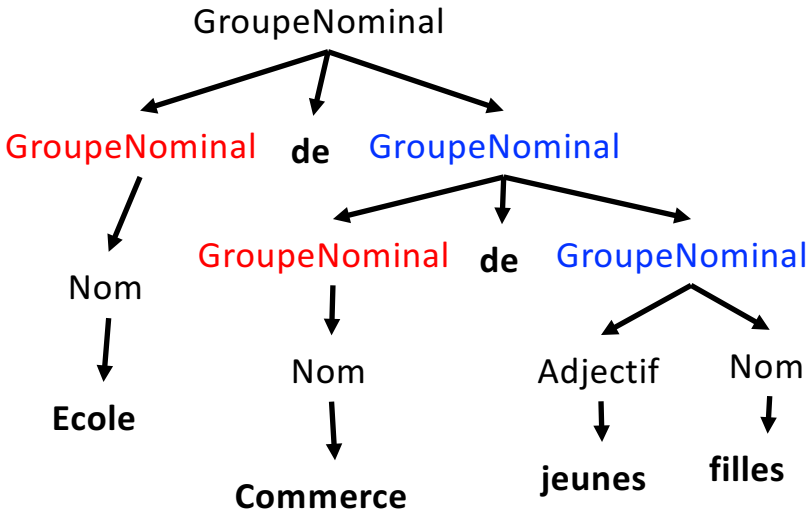
Grammaire ambigüe

Exemple

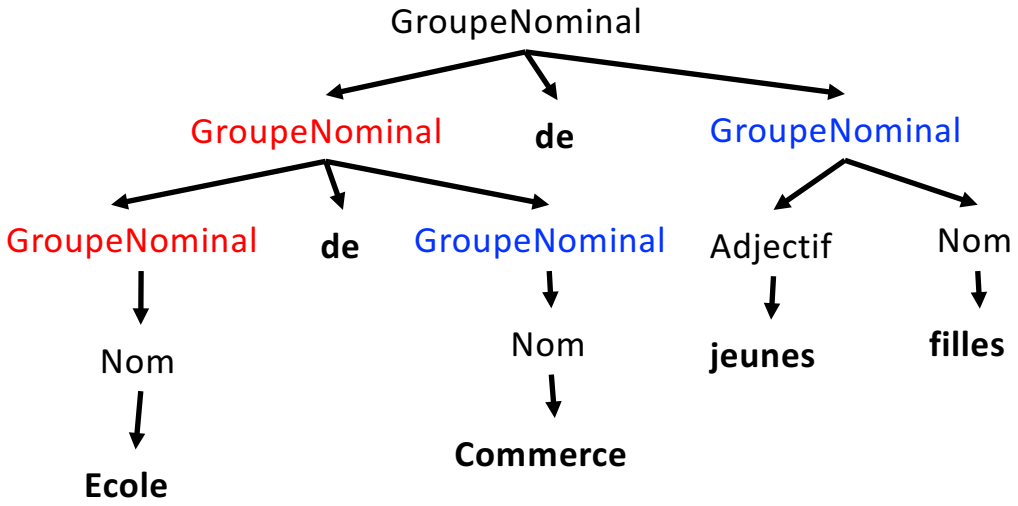
Ecole de commerce de filles jeunes



Grammaire ambigüe
Exemple



Ecole de commerce de jeunes filles



Ecole de commerce de filles jeunes

2 arbres syntaxiques différents d'une expression (i.e. **ambigüité de la grammaire**) ⇒ **2** interprétations différentes ⇒ **2** sémantiques différentes ⇒ **2** codes possibles à générer pour cette expression ⇒ **Problème de non-déterminisme pour le compilateur !!**

Grammaire ambigüe
Exercice 1

La grammaire suivante est-elle ambiguë ?

G_{arith}:

r1 : $E \rightarrow E \times E$

r2 : $E \rightarrow E + E$

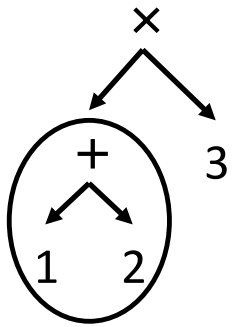
r3 : $E \rightarrow$ Nombre

Si Oui Pourquoi ?

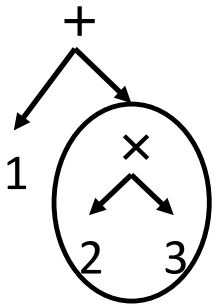
Grammaire ambigüe
Solution

- Le mot "$1 + 2 \times 3$" possède deux dérivations droites
  - $E \Rightarrow_{r1} E \times E \Rightarrow_{r2} E + E \times E \Rightarrow^*_{r3} 1 + 2 \times 3$
  - $E \Rightarrow_{r2} E + E \Rightarrow_{r1} E + E \times E \Rightarrow^*_{r3} 1 + 2 \times 3$
- Donc 2 arbres syntaxiques

Arbre 1



Arbre 2



- Donc 2 interprétations possibles
  - **$(1 + 2) \times 3 \equiv 9$**
  - **$1 + (2 \times 3) \equiv 7$**

Grammaire ambigüe
Exercice

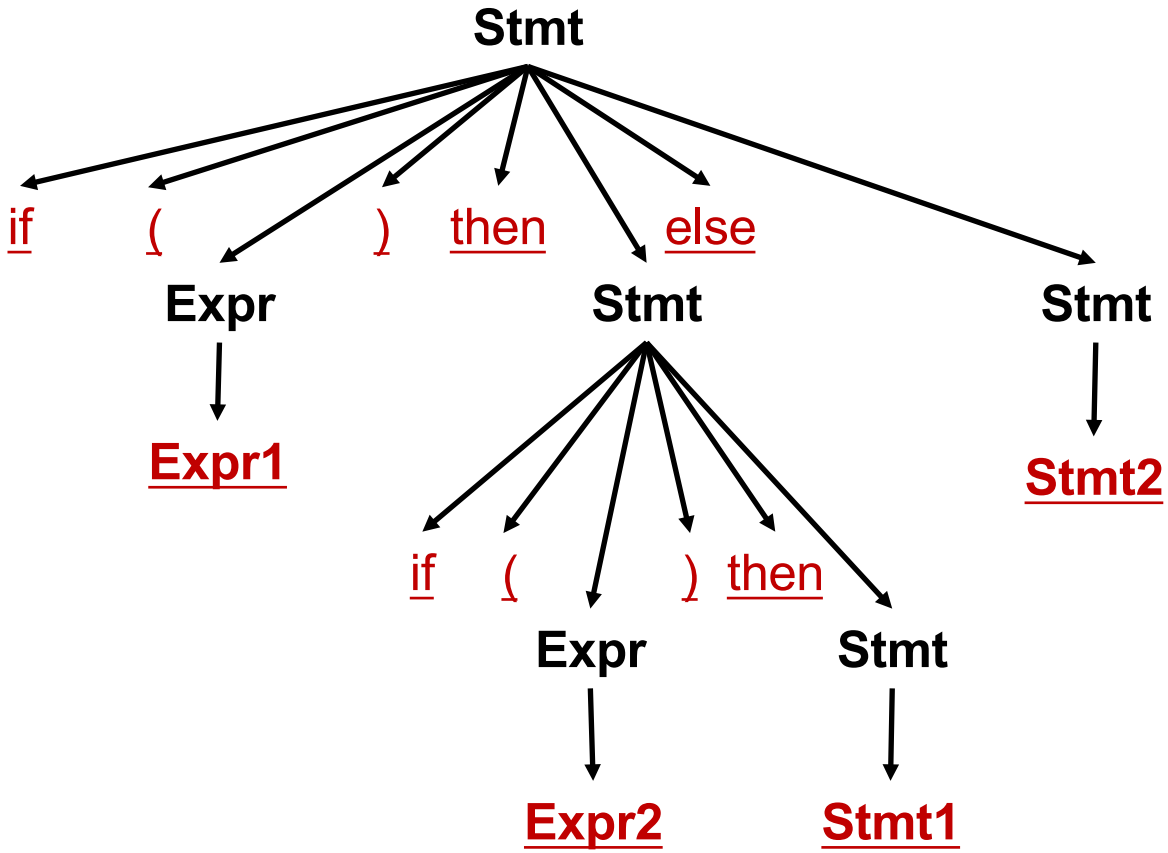
- *G:*
  - r1: **Stmt** → if (**Expr**) then **Stmt** else **Stmt**
  - r2: | if (**Expr**) then **Stmt**
  - …
  - rn | . . .

  - Trouver un arbre de dérivation du mot w = if (*Expr*1) then if (*Expr*2) then *Stmt*1 else *Stmt*2

Grammaire ambigüe

Solution 1

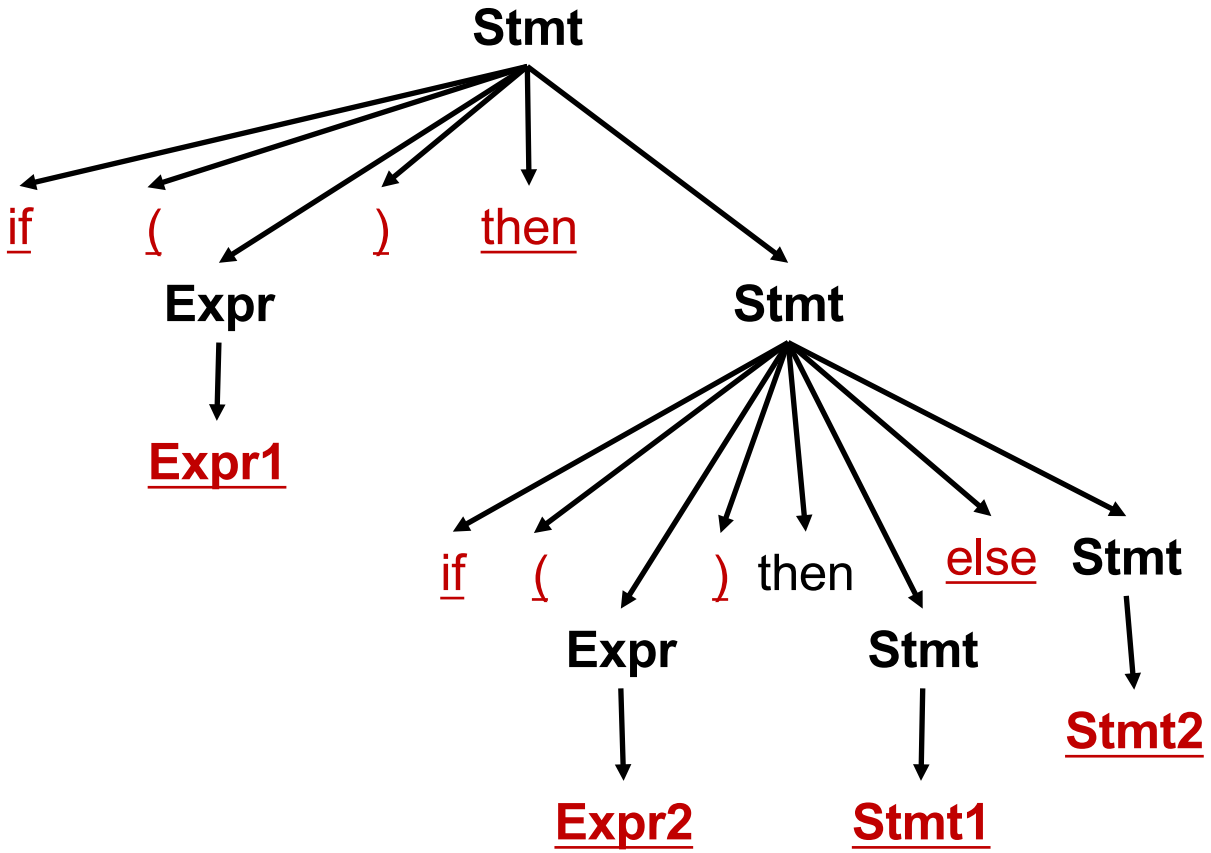
- Stmt $\Rightarrow^{r1}$
- if (**Expr**) then
  - **Stmt**
- else **Stmt**
- $\Rightarrow^{r2}$
- if (**Expr**) then
  - if (**Expr**) then
    - **Stmt**
- else **Stmt**
- …
- $\Rightarrow$
- if (*Expr*1) then
  - if (*Expr2*) then
    - *Stmt*1
- else
  - *Stmt2*



Grammaire ambigüe

Solution 2

- Stmt $\Rightarrow^{r2}$
- if (**Expr**) then
  - **Stmt**
- $\Rightarrow^{r1}$
- if (**Expr**) then
  - if (**Expr**) then
    - **Stmt**
  - else **Stmt**
- …
- $\Rightarrow$
- if (*Expr*1) then
  - if (*Expr*2) then
    - *Stmt*1
  - else
    - *Stmt*2



G est une grammaire ambiguë du fait qu'on a trouvé deux dérivations du mot w = « if (*Expr*1) then if (*Expr*2) then *Stmt*1 else *Stmt*2 »

Grammaire ambigüe

- L'ambiguïté implique que l'analyseur syntaxique ne pourra pas découvrir d'une manière unique et définitive l'arbre syntaxique de cette expression.

- Si l'analyseur syntaxique ne peut pas décider la structure syntaxique d'une expression, décider du sens (i.e. la sémantique) et donc du code exécutable équivalent à cette expression ne sera pas possible !!

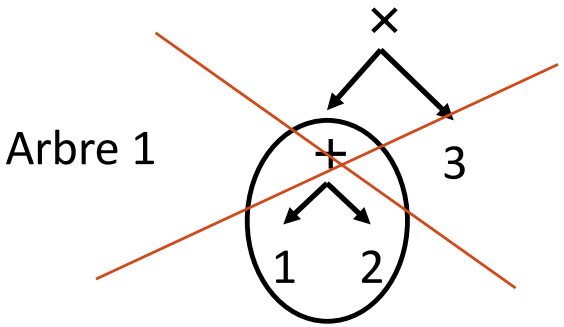
- L'Ambiguïté est donc une propriété indésirable dans une grammaire.

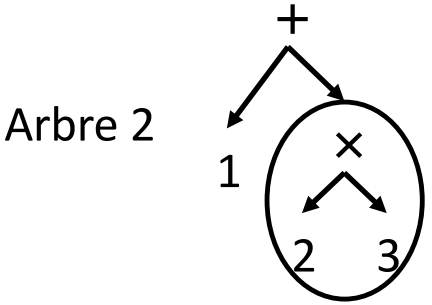
Grammaire ambigüe

- Lorsque le langage définit des strings composés d'instructions et d'opération, l'arbre syntaxique (qui va déterminer le code produit par le compilateur) doit refléter :
  - Les priorités
  - Les associativités

Grammaire ambigüe
Exemple

- Le mot "$1 + 2 \times 3$" possède deux dérivations droites
  - ~~$E \Rightarrow_{r1} E \times E \Rightarrow_{r2} E + E \times E \Rightarrow^*_{r3} 1 + 2 \times 3$~~
  - $E \Rightarrow_{r2} E + E \Rightarrow_{r1} E + E \times E \Rightarrow^*_{r3} 1 + 2 \times 3$
- Donc 2 arbres syntaxiques





- Donc 2 interprétations possibles
  - ~~**(1 + 2) × 3 ≡ 9**~~
  - **1 + (2 × 3) ≡ 7**

C'est la 2ème interprétation qui correspond aux règles de priorité usuelles des opérateurs arithmétiques !!

Grammaire ambigüe

Elimination de l'ambiguïté

- Ambiguïté pour les expressions de G_{arith} sur l'opérateur $\times$
  - Cause : G_{arith} accepte aussi bien l'associativité gauche que droite de $\times$
  - Solution : On choisit d'accepter ***l'associativité à gauche de $\times$***
    (e.g. l'interprétation $(1 \times 2) \times 3$ mais pas $1 \times (2 \times 3)$.)

- Ambiguïté pour les expressions de G_{arith} sur l'opérateur $+$
  - Cause : G_{arith} accepte aussi bien l'associativité gauche que droite de $+$
  - Solution : On choisit d'accepter ***l'associativité à gauche de $+$***
    (e.g. l'interprétation $(1 + 2) + 3$ mais pas $1 + (2 + 3)$.)

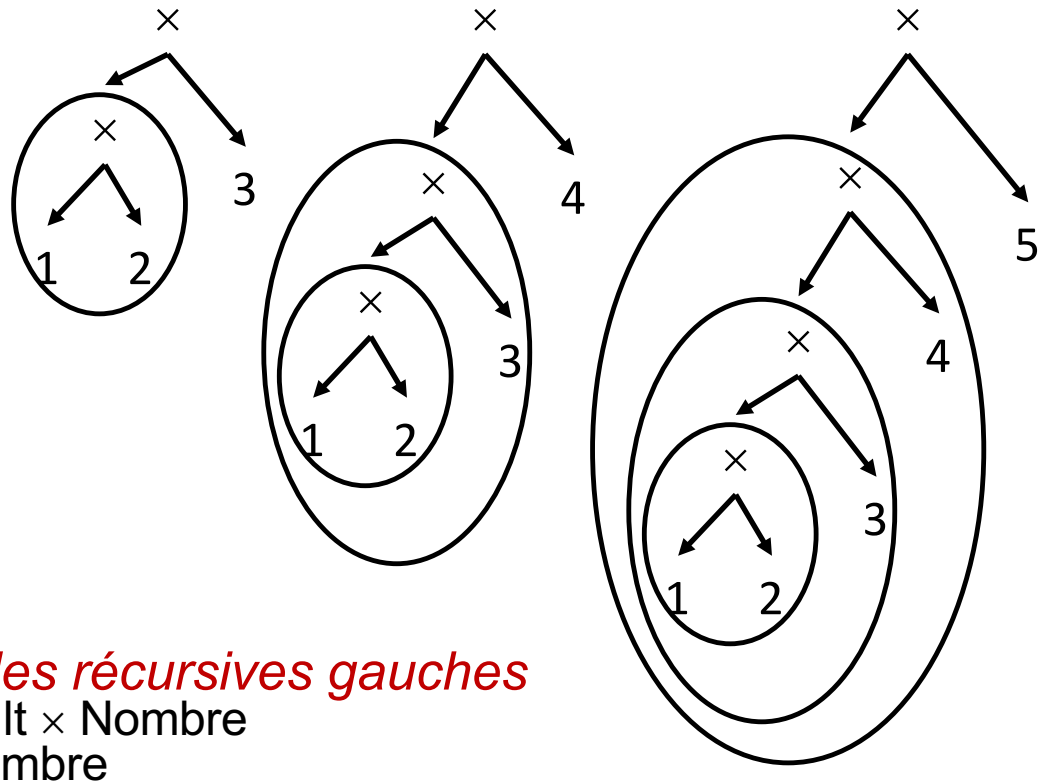
- Ambiguïté pour les expressions de G_{arith} sur les 2 opérateurs $\times$ *et* $+$
  - Cause : G_{arith} ne différencie pas entre les priorités de $\times$ et $+$
  - Solution : On choisit ***priorité($\times$) > priorité($+$)***
    (e.g. l'interprétation $1 + (2 \times 3)$ mais pas $(1 + 2) \times 3$.)

- NB. Rien ne nous empêche d'adopter l'associativité droite !!
- Suite de l'exercice :
  - Essayer de réfléchir sur les transformations de G_{arith} pour supporter ces contraintes nécessaires à sa désambiguïsation

Grammaire ambigüe

Elimination de l'ambiguïté

- Accepter *l'associativité à gauche de* $\times$



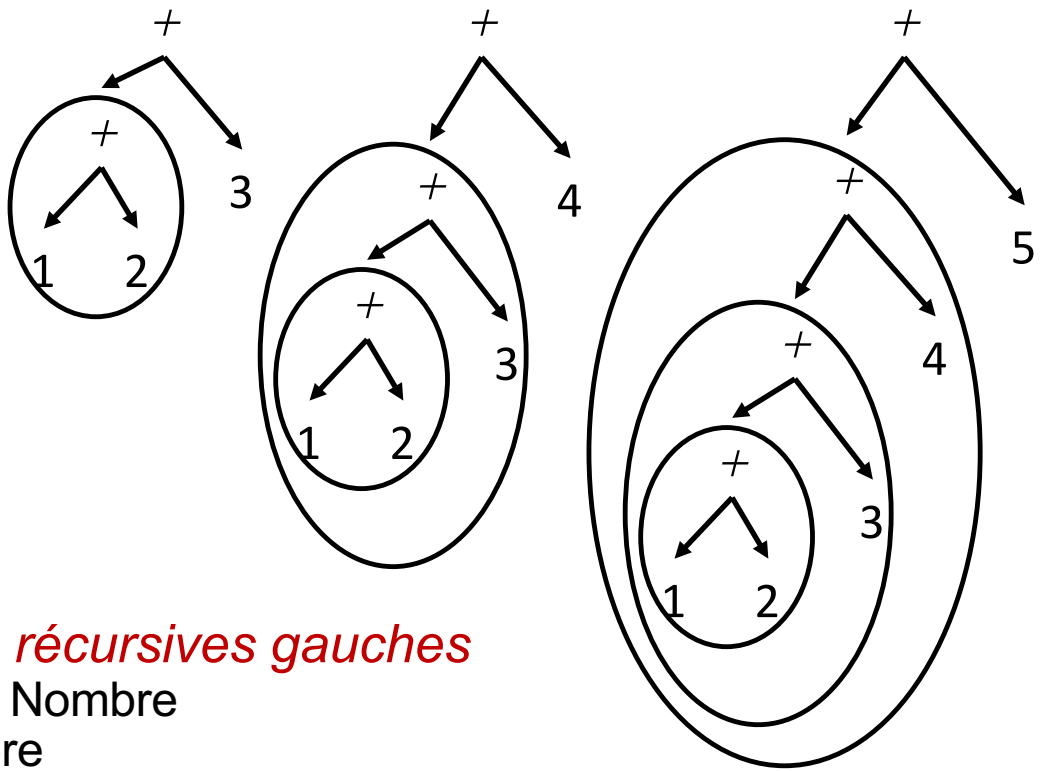
$\Rightarrow$ Introduire des *règles récursives gauches*

- R_{11} : Mult $\rightarrow$ Mult $\times$ Nombre
- R_{12} : Mult $\rightarrow$ Nombre

Grammaire ambigüe

Elimination de l'ambiguïté

- Accepter *l'associativité à gauche de +*



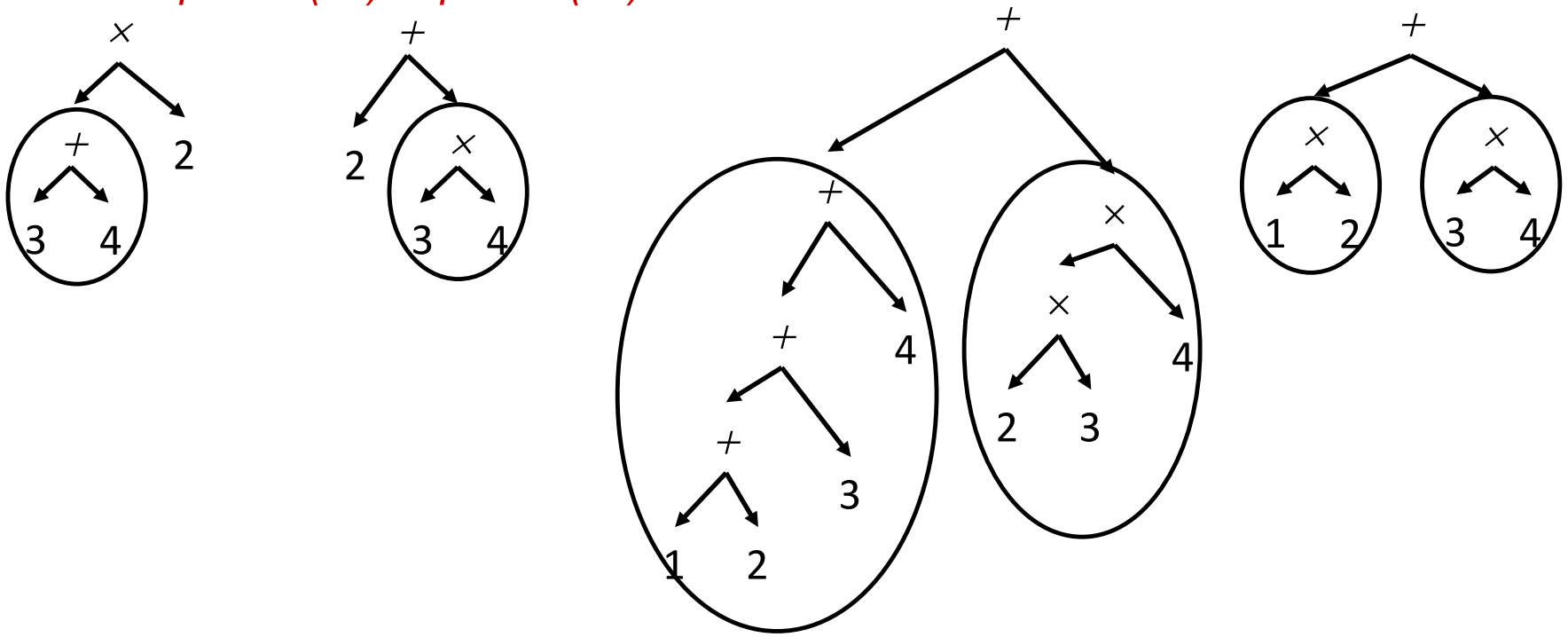
⇒ Introduire des *règles récursives gauches*

- R_{21} : Add → Add + Nombre
- R_{22} : Add → Nombre

Grammaire ambigüe

Elimination de l'ambiguïté

- Choisir *priorité(×) > priorité(+)*



⇒ Introduire des *règles récursives gauches*

- R_{31} : Add → Add + Mult (*plus générale que* R_{21}, *car* R_{12} : Mult → Nombre)
- R_{32} : Add → Mult (*plus générale que* R_{22}, *car* R_{12} : Mult → Nombre)

Grammaire ambigüe

Elimination de l'ambiguïté

- G'_{arith} < T={Nombre,+, ×}, NT ={Mult, Add}, S=Add, P = {R_{11}, R_{12}, R_{31}, R_{32}} >
  - R_{11} : Mult → Mult × Nombre
  - R_{12} : Mult → Nombre
  - ~~R_{21} : Add → Add + Nombre~~ (car *R_{31} est plus générale que R_{21}*)
  - ~~R_{22} : Add → Nombre~~ (car *R_{32} est plus générale que R_{22}*)
  - R_{31} : Add → Add + Mult
  - R_{32} : Add → Mult

- On a pris *S=Add* car *Add* est plus général de *Mult* (c.f. R_{32}, de *Add* on peut aller vers *Mult* mais l'inverse est faux)

  - G'_{arith} n'est pas ambiguë par construction !!

Grammaire ambigüe
Elimination de l'ambiguïté

- Défaut de G'_{arith}
  - C'est impossible de forcer
    - 1 + 2 × 3 à être interprété (1 + 2) × 3
    - et 3 × 1 + 2 à être interprété 3 × (1 + 2)

- Suite de l'exercice :
  - Penser à une solution à ces nouveaux problèmes

Grammaire ambigüe

Elimination de l'ambiguïté

- Solution possible : introduire les parenthèses
  - NT devient {Mult, Add, Aux}
  - P devient {R'_{11}, R'_{12}, R'_{13}, R'_{14}, R_{31}, R_{32}}
    - R'_{11} : Mult $\rightarrow$ Mult $\times$ Aux *(plus générale que R_{11}, car* R'_{13} : Aux$\rightarrow$ Nombre*)*
    - R'_{12} : Mult $\rightarrow$ Aux *(plus générale que R_{12}, car* R'_{13} : Aux$\rightarrow$ Nombre*)*
    - R'_{13} : Aux $\rightarrow$ Nombre
    - R'_{14} : Aux $\rightarrow$ (Add)

    - Au lieu de
    - R_{11} : ~~Mult $\rightarrow$ Mult $\times$ Nombre~~
    - R_{12} : ~~Mult $\rightarrow$ Nombre~~

Grammaire ambigüe

Elimination de l'ambiguïté

- 4 cas avec deux opérations dans une expression parenthèsées
  - + à gauche du × : x + y + z
    - Cas particuliers : Nombre + Add, Add + Nombre
  - × à gauche du + : x + y × z
    - Exemples : (1 + 2) × 3
    - Cas particuliers : Nombre + Mult, Add × Nombre
  - + à gauche du + : x × y + z
    - Exemples : 3 × (1 + 2)
    - Cas particuliers : Nombre × Add, Mult + Nombre
  - × à gauche du × : x × y × z
    - Cas particuliers : Mult × Nombre, Nombre × Mult
- Seuls l'associativité gauche sera bien sûr prise en compte

Grammaire ambigüe

Elimination de l'ambiguïté

- $G'_{arith} < T= \{Nombre,+, \times\}, NT = \{Mult, Add, Aux\}, S = Add, P = \{R_1, R_2, R_3, R_4, R_5, R_6\}>$
  - R1 (R'_{11}) : Mult → Mult × Aux
  - R2 (R'_{12}) : Mult → Aux
  - R3 (R'_{13}) : Aux → Nombre
  - R4 (R'_{14}) : Aux → (Add)
  - R5 (R_{31}) : Add → Add + Mult
  - R6 (R_{32}) : Add → Mult

    - 1 + 2 × 3 pourra être interprété sans ambiguïté (1 + 2) × 3 :
      - Add $\Rightarrow_{R6}$ Mult $\Rightarrow_{R1}$ Mult × Aux $\Rightarrow_{R2}$ Aux × Aux $\Rightarrow_{R4}$ (Add) × Aux $\Rightarrow_{R5}$ (Add + Mult) × Aux $\Rightarrow_{R6}$ (Mult + Mult) × Aux $\Rightarrow_{R2}$ (Aux + Mult) × Aux $\Rightarrow_{R2}$ (Nombre + Mult) × Aux $\Rightarrow^*$ (Nombre + Nombre) × Nombre

    - 3 × 1 + 2 peut être interprété sans ambiguïté 3 × (1 + 2)
      - Add $\Rightarrow_{R6}$ Mult $\Rightarrow_{R1}$ Mult × Aux $\Rightarrow_{R2}$ Aux × Aux $\Rightarrow_{R3}$ Nombre × Aux $\Rightarrow_{R4}$ Nombre × (Add) $\Rightarrow^*$ Nombre × (Nombre + Nombre)

Analyseur Syntaxique
Types

- Il existe 2 types d'analyseurs syntaxiques (parseurs)
  - Analyseur descendant (top-down)
  - Analyseurs ascendants (bottom-up)

Analyseur Syntaxique
Types

- Analyseur descendant (top-down)
  - Algorithme :
    - Commence par la racine et procède en descendant l'arbre syntaxique jusqu'aux feuilles.
    - A chaque étape, l'analyseur choisit un nœud parmi les symboles non-terminaux et développe l'arbre à partir de ce nœud.
  - Exemples : Analyseur récursif descendant, Analyseur LL(1)

Analyseur Syntaxique
Types

- Analyseurs ascendants (bottom-up)
  - Algorithme :
    - Commence par les feuilles et procède en remontant l'arbre syntaxique jusqu'à la racine.
    - A chaque étape, l'analyseur ajoute des nœuds qui développent l'arbre partiellement construit.
  - Exemples : Analyseur LR(1), Analyseur LALR(1), Analyseur SLR(1)
    - LR(1) $\supset$ LALR(1) $\supset$ SLR(1)

Analyseur Syntaxique Descendant (top-down)

Analyseur Syntaxique Descendant Exemple

- Pour une grammaire du type
  - R1 : S $\rightarrow$ Mult
  - R2 : Mult $\rightarrow$ Mult $\times$ Aux
  - R3 : Mult $\rightarrow$ Aux
  - R4 : Aux $\rightarrow$ Nombre

- Problème :
  - L'algorithme risque de **développer** Mult en Mult $\times$ Aux puis en Mult $\times$ Mult $\times$ Aux puis en Mult $\times$ Aux $\times$ Aux $\times$ Aux … $\times$ Aux $\times$ Aux **indéfiniment** : **bouclage de l'analyseur** !!

- Solution :
  - **Éliminer la récursivité gauche** de la grammaire pour une analyse top-down

Analyseur Syntaxique Descendant
Récursivité a gauche

- Une règle de réécriture est dite récursive gauche si
  - Le premier symbole sur la partie droite de la règle est le même que celui sur sa partie gauche
    - Exemple :
      - $S \rightarrow Sa$
  - ou si le symbole de la partie gauche de la règle apparaît sur sa partie droite et tous les symboles qui le précèdent peuvent dériver le mot vide
    - Exemple :
      - $S \rightarrow TSa$
      - $T \rightarrow \varepsilon \mid \ldots$

Analyseur Syntaxique Descendant

- ***G : Grammaire récursive gauche (NT = {S})***
  - ***S*** → ***S*** α *(i.e. tout mot de L(G) se termine pas une suite de α)*
  - | β *(i.e. tout mot de L(G) commence par β ou contient seulement β)*
  - $S \Rightarrow S\,\alpha \Rightarrow S\,\alpha\,\alpha \Rightarrow \ldots \Rightarrow S\,\alpha^* \Rightarrow$ ***β α****
  - ***L(G') = β α****
- ***G' : Grammaire non-récursive gauche équivalente à G (NT' = {S,R})***
  - ***S*** → β ***R*** *(i.e. tout mot de L(G') commence par β)*
  - ***R*** → α ***R*** *(i.e. tout mot de L(G') se termine éventuellement pas une suite de α)*
  - | ε
  - $S \Rightarrow \beta\,R \Rightarrow \beta\,\alpha\,R \Rightarrow \alpha\,\alpha\,R \Rightarrow \ldots \Rightarrow \alpha^*\,R \Rightarrow$ ***β α****
  - ***L(G') = β α****
- L(G) = L(G') donc la transformation de G en G' a conservé le langage reconnu

Elimination de la récursivité a gauche Solution

Entrée: Une grammaire G.

Sortie: Une grammaire équivalente sans récursivité à gauche.

Méthode:

Ordonner les non-terminaux par indices A_1, ..., A_n.

Pour i:=1 à n faire

Pour j : = 1 à i - 1 faire

Remplacer chaque production de la forme $A_i \rightarrow A_j\gamma$

par les productions $A_i \rightarrow \delta_1\gamma \mid ... \mid \delta_k\gamma$,

où $A_j \rightarrow \delta_1 \mid . . . \mid \delta_k$ sont les A_j -productions actuelles

Fin

Eliminer les récursivité immédiates à gauche des A_i-productions

Fin

Elimination de la récursivité a gauche Solution

- Éliminer la récursivité gauche de la grammaire suivante :
  - G'_{arith} < T={Nombre,+, ×}, NT={Mult, Add, Aux}, S=Add, P = {R_1, R_2, R_3, R_4, R_5, R_6}>
    - R1 : Mult $\rightarrow$ Mult $\times$ Aux
    - R2 : Mult $\rightarrow$ Aux
    - R3 : Aux $\rightarrow$ Nombre
    - R4 : Aux $\rightarrow$ (Add)
    - R5 : Add $\rightarrow$ Add + Mult
    - R6 : Add $\rightarrow$ Mult

Elimination de la récursivité a gauche Solution

Règles récursives gauche **NT={Mult, Aux, Add}**	**Grammaire récursive droite équivalente** **(i.e. conserve l'associativité droite et les priorités de G'_{arith})** **NT={Mult, Mult', Aux, Add, Add'}**
	// **Transformation directe de Mult**
R_1 : Mult $\rightarrow$ Mult $\times$ Aux	Mult $\rightarrow$ Aux Mult'
R_2 : Mult $\rightarrow$ Aux	
	Mult' $\rightarrow$ $\times$ Aux Mult'
	| ε
	// **Aux n'est pas récursive gauche**
R_3 : Aux $\rightarrow$ Nombre	Aux $\rightarrow$ Nombre
R_4 : Aux $\rightarrow$ (Add)	Aux $\rightarrow$ (Add)
	// **Transformation directe de Add**
R_5 : Add $\rightarrow$ Add $+$ Mult	Add $\rightarrow$ Mult Add'
R_6 : Add $\rightarrow$ Mult	Add' $\rightarrow$ $+$ Mult Add'
	| ε

Elimination de la récursivité a gauche Solution

- Éliminer la récursivité gauche de la grammaire suivante :

- $S \rightarrow S + A \mid S \times B \mid C \mid D$

Elimination de la récursivité a gauche Solution

$S \rightarrow S\,\alpha_1 \mid S\,\alpha_2 \mid S\,\alpha_3 \mid \ldots \mid S\,\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$ ➔ $S \rightarrow \beta_1\,R \mid \beta_2\,R \mid \ldots \mid \beta_n\,R$, $R \rightarrow \alpha_1\,R \mid \alpha_2\,R \mid \ldots \mid \alpha_n\,R \mid \varepsilon$

- $S \rightarrow S + A \mid S \times B \mid C \mid D$

- $S \rightarrow C\ S' \mid D\ S'$
- $S' \rightarrow + A\ S' \mid \times B\ S' \mid \varepsilon$

Factorisation à gauche
Algorithme

- *Pour chaque $A \in NT$*
  - *Trouver le plus long préfixe « α » commun à deux ou plusieurs parties droites de règles relatives à A*
  - *Si ($\alpha \neq \varepsilon$) alors*
    - *remplacer les règles de A de la forme*
      - $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$
    - *par :*
      - $A \rightarrow \alpha\ B \mid \gamma$
      - $B \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$
      - *$NT \leftarrow NT \cup \{B\}$ // ajouter B à l'ensemble des symboles non-terminaux*
- *Répéter jusqu'à ce qu'il n'y ait plus de préfixe commun*

Factorisation à gauche
Exemple

- $S \rightarrow aS \mid aB \mid aC \mid aD$

Simple factorisation

- $S \rightarrow a\ S'$
- $S' \rightarrow S \mid B \mid C \mid D$

Factorisation à gauche
Exercice

Factorisation à gauche de:

- S → *aA
- |*aB
- |*C

Factorisation à gauche
Solution

- S → *aA | *aB | *C

  - S → *aS’ | *C
  - S’ → A | B

    - S → *S’’
    - S’’ → aS’ | C

Analyseur Syntaxique Descendant
Grammaire prédictive LL(1)

Analyseur Syntaxique Descendant

Grammaire prédictive LL(1)

LL : Left to right – Leftmost derivation

- Commence par la racine et procède en descendant l'arbre syntaxique jusqu'aux feuilles.

- A chaque étape, l'analyseur choisit un nœud parmi les symboles non-terminaux et développe l'arbre à partir de ce nœud.

Analyseur Syntaxique Descendant

Grammaire prédictive LL(1)

LL : Left to right – Leftmost derivation

- Une grammaire est dite **LL(k)** si, et seulement si, elle peut être analysée en ne disposant, à chaque instant, que des **n** prochains **terminaux** non encore consommés.
- Le **k** appelé lookahead (regarder en avant) indique le nombre de **terminaux** qu'il faut avoir lus sans les avoir encore consommés pour décider quelle dérivation faire.
- L'analyse d'une grammaire **LL(3)** impose de gérer 3 variables contenant les 3 prochains **terminaux** non encore consommés à chaque instant.

Analyseur Syntaxique Descendant

Grammaire prédictive LL(1)
LL : Left to right – Leftmost derivation

- Soit $A \rightarrow \alpha$ une règle de production, on définit **First(α)** =
  - {ensemble des symboles terminaux qui peuvent apparaître comme premiers symboles dans les mots dérivables à partir de α}

- Soit $G = <T, NT, S, P>$ une grammaire, G est dite prédictive (dite LL(1)) ssi
- $\forall$ les règles $A \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \cdots \alpha_n$ de **P**
  - $\forall$ i,j $First(\alpha_i) \cap First(\alpha_j) = \varnothing$

Analyseur Syntaxique Descendant

Grammaire prédictive LL(1)

Exemple

- La grammaire S → a | bS est **LL(1)**:
  - Productions(S→a) = First(a) = {a}
  - Productions(S→ bS) = First(bS) = First (b) = {b}

- La grammaire S → a | aS n'est pas **LL(1)**:
  - Productions(S→a) = First (a) = {a}
  - Productions(S→ aS) = First (aS) = First (a) = {a}

Analyseur Syntaxique Descendant

Grammaire prédictive LL(1)
First/Follow

- L'ensemble premier :

Productions(S→a) = **Premier**(a) si a n'est pas vide.

- Si a est de la forme aS ou a est un **terminal** alors nous avons **First**(aS) = {a}.

- Si a est de la forme AS ou A est un **non-terminal** alors nous avons **First**(AS) = **First**(A).

Analyseur Syntaxique Descendant

Grammaire prédictive LL(1)

First/Follow

- L'ensemble suivant :

Productions(S→a) = **Follow**(S) si a est vide.

- Si une règle est de la forme S → aTb alors l'ensemble **Follow**(T) contient l'ensemble **First**(b).
- Si une règle est de la forme A→ aT alors l'ensemble **Follow**(T) contient l'ensemble **Follow**(A).

Analyseur Syntaxique Descendant

Grammaire prédictive LL(1)

Exercice : First

Calculer l'ensemble First (S) des règles suivantes :

- $S \rightarrow Aa$
- $A \rightarrow bB$
- $\quad\quad | \varepsilon$

Calculer l'ensemble First des non terminaux de la grammaire suivante :

- $S \rightarrow ETC$
- $E \rightarrow aE \mid \varepsilon$
- $T \rightarrow bT \mid cT \mid \varepsilon$
- $C \rightarrow dC \mid da \mid dE$

Analyseur Syntaxique Descendant

Grammaire prédictive LL(1)

Solution: First

- $S \rightarrow Aa$
- $A \rightarrow bB$
- $\quad\quad | \ \varepsilon$

First (S) = First (A) + First(S si A=ε) = {b,a}

- $S \rightarrow ETC$
- $E \rightarrow aE \mid \varepsilon$
- $T \rightarrow bT \mid cT \mid \varepsilon$
- $C \rightarrow dC \mid da \mid dE$

First (S) = {a,b,c,d}

First (E) = {a,ε}

First(T) = {b,c,ε}

First(C) = {d}

Analyseur Syntaxique Descendant

Grammaire prédictive LL(1)

Exercice : Follow

Calculer l'ensemble Follow des non terminaux de la grammaire suivante :

- $S \rightarrow ETC$
- $E \rightarrow aE \mid \varepsilon$
- $T \rightarrow bT \mid cT \mid \varepsilon$
- $C \rightarrow dC \mid da \mid dE$

Analyseur Syntaxique Descendant

Grammaire prédictive LL(1)

Solution: Follow

- $S \rightarrow ETC$
- $E \rightarrow aE \mid \varepsilon$
- $T \rightarrow bT \mid cT \mid \varepsilon$
- $C \rightarrow dC \mid da \mid dE$

First (S) = {a,b,c,d}

First (E) = {a,ε}

First(T) = {b,c,ε}

First(C) = {d}

Follow (S) = {$}

Folow (E) = First(T)+First(C)={b,c,d,$}

Follow(T) = First(C) = {d}

Follow (C) = Follow(S) = {$}

Analyseur Syntaxique Descendant
Grammaire prédictive LL(1)
Exemple

- Déterminons un programme permettant une analyse syntaxique en reprenant la grammaire G suivantes:
  - expr ➔ term
  - | term “+“ expr
  - | term “-“ expr
  - term ➔ fact
  - | fact “*“ term
  - | fact “/“ term
  - fact ➔ “a“ | “b“ | “c“ | “d“
  - | (“expr“)
  - | fact “^“fact
  - expr /*axiome*/)
- Nous décrirons seulement les règles de productions de fact.

Analyseur Syntaxique Descendant

Grammaire prédictive LL(1) Exemple

- fact ➔ “a“ | “b“ | “c“ | “d“
- | (“expr“)
- | fact “^“fact

Pour la fonction fact, nous obtenons alors :

```
void _fact()
{
 if (sTerminal == “ ( “) {  /*commençons par “(“ expr “)“ */
 _expr();
 if (sTerminal != “ ) “)
      {_ErrSyntax("après expr “ ) “ attendu");}
 else
      {_avancer();}
 } /*fin de “(“ expr “)“ */
else
 {
```

Analyseur Syntaxique Descendant

Grammaire prédictive LL(1)

Exemple

- fact ➔ “a“ | “b“ | “c“ | “d“
- | (“expr“)
- | fact “^“fact

```
{
  if (sTerminal == (“a“ || “b“ || “c“ || “d“) {
      _avancer()
   }
  else
   {
     _fact();
     if (sTerminal != “^“)
           {_ErrSyntax(‘’après FACT “^“ attendu ’’);}
     else {
        _fact();
        _avancer();
        } /*fin de fact “^“ fact*/
   }
} /*Fin de la fonction pour les règles de fact*/
```

Analyseur Syntaxique Descendant
Grammaire prédictive LL(1)

- Nous venons de produire un programme permettant de construire uniquement une analyse de la production Fact. Mais pouvons-nous être plus général pour l'ensemble des constructions ?

- L'**analyse prédictive** se fait à l'aide d'une **table d'analyse syntaxique prédictive**.

Analyseur Syntaxique Descendant

Grammaire prédictive LL(1)

Table d'Analyse

First (S) = {a,b,c,d} Follow (S) = {$}

First (E) = {a,ε} Folow (E) = First(T)+First(C)={b,c,d,$}

First(T) = {b,c,ε} Follow(T) = First(T) = {d}

First(C) = {d} Follow (C) = Follow(S) = {$}

	First	Follow
S	{a,b,c,d}	{$}
E	{a,ε}	{b,c,d,$}
T	{b,c,ε}	{d}
C	{d}	{$}

Analyseur Syntaxique Descendant

Grammaire prédictive LL(1)

Table d'Analyse

- $S \rightarrow ETC$
- $E \rightarrow aE \mid \varepsilon$
- $T \rightarrow bT \mid cT \mid \varepsilon$
- $C \rightarrow dC \mid da \mid dE$

$C \rightarrow dC'$

$C' \rightarrow C \mid a \mid E$

	First	Follow
S	{a,b,c,d}	{$}
E	{a,ε}	{b,c,d,$}
T	{b,c,ε}	{d}
C	{d}	{$}

	a	**b**	**c**	**d**	$
S	$S \rightarrow ETC$	$S \rightarrow ETC$	$S \rightarrow ETC$	$S \rightarrow ETC$	
E	$E \rightarrow aE$	$E \rightarrow \varepsilon$	$E \rightarrow \varepsilon$	$E \rightarrow \varepsilon$	$E \rightarrow \varepsilon$
T		$T \rightarrow bT$	$T \rightarrow cT$	$T \rightarrow \varepsilon$	
C				$C \rightarrow dC$ $C \rightarrow da$ $C \rightarrow dE$	

Table d'Analyse

N'est pas une grammaire LL(1)

Factorisation

Analyseur Syntaxique Descendant
Grammaire prédictive LL(1)
Exercice

Soit la grammaire suivante :

- $S \rightarrow S + T \mid T$
- $T \rightarrow T*F \mid F$
- $F \rightarrow (S) \mid id$

Construire la table d'analyse

Analyseur Syntaxique Descendant
Grammaire prédictive LL(1)
Solution

Etape 1: Supprimer la récursivité a gauche

- $S \rightarrow S + T \mid T$
- $T \rightarrow T*F \mid F$
- $F \rightarrow (S) \mid id$

- $S \rightarrow TS'$
- $S' \rightarrow + T S' \mid \varepsilon$
- $T \rightarrow FT'$
- $T' \rightarrow *FT' \mid \varepsilon$
- $F \rightarrow (S) \mid id$

Nouvelle grammaire

Analyseur Syntaxique Descendant
Grammaire prédictive LL(1)
Solution

Calculer l'ensemble First et Follow

- $S \rightarrow TS'$
- $S' \rightarrow + T S' \mid \varepsilon$
- $T \rightarrow FT'$
- $T' \rightarrow *FT' \mid \varepsilon$
- $F \rightarrow (S) \mid id$

	First	Follow
S	{ (, id }	{) , $ }
S'	{ + , ε}	{) , $ }
T	{ (, id }	{ + ,) , $ }
T'	{ * , ε}	{ + ,) , $ }
F	{ (, id }	{ * , + ,) , $ }

Analyseur Syntaxique Descendant

Grammaire prédictive LL(1)
Solution

- $S \rightarrow TS'$
- $S' \rightarrow + T S' \mid \varepsilon$
- $T \rightarrow FT'$
- $T' \rightarrow *FT' \mid \varepsilon$
- $F \rightarrow (S) \mid id$

	First	Follow
S	{ (, id }	{) , $ }
S'	{ + , ε}	{) , $ }
T	{ (, id }	{ + ,) , $ }
T'	{ * , ε}	{ + ,) , $ }
F	{ (, id }	{ * , + ,) , $ }

Table d'analyse

	id	**+**	*****	**(**	**)**	**$**
S	$S \rightarrow TS'$			$S \rightarrow TS'$		
S'		$S' \rightarrow + T S'$			$S' \rightarrow \varepsilon$	$S' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (S)$		

Analyseur Syntaxique Descendant

Grammaire prédictive LL(1)

Exemple: Vérification de mot

- Prenons la grammaire G suivante :
  - S → Term Exp
  - Exp → +Term Exp | ε
  - Term → Entier ou “0“ | “1“ | .. | “9“ | ..

Analyseur Syntaxique Descendant

Grammaire prédictive LL(1)

Exemple: Vérification de mot

- S → Term Exp
- Exp → +Term Exp | ε
- Term → Entier ou "0" | "1" | .. | "9" | ..

	First	Follow
S	{ entier }	{ $ }
Exp	{+ , ε}	{ $ }
Term	{ entier }	{+ , $ }

Table d'analyse

	+	entier	$
S		S → Term Exp	
Exp	Exp → +Term Exp		Exp → ε
Term		Term → Entier	

Analyseur Syntaxique Descendant

Grammaire prédictive LL(1)

Exemple: Vérification de mot

Le mot :1+3

Reconnu	Pile	Entrée	Action
	S$	1 + 3$	S → Term Exp
	Term Exp$	1 + 3$	Term → Entier
	Entier Expr$	1 + 3$	Entier = 1 (dépilé Entier)
1	Expr$	+ 3$	Exp → +Term Exp
1	+Term Exp$	+ 3$	+=+ (Dépilé +)
1+	Term Exp$	3$	Term → Entier
1+	Entier Expr$	3$	Entier = 3 (dépilé Entier)
1+3	Expr$	$	Exp → ε
	$	$	Accepté

Exercice 1

- On s'intéresse à une grammaire GI de calcul d'itinéraire suivante:
  - G = <T,NT, {ROUTE}, P> avec
  - T = { go, tg, td, pan },
  - NT = {ROUTE, INST, PANNEAU , TOURNE}
  - Les règles P
    - ✓ ROUTE→ INST| INST ROUTE
    - ✓ INST→ go| PANNEAU TOURNE
    - ✓ TOURNE→ tg| td
    - ✓ PANNEAU→ ε | pan

1. Cette grammaire n'est pas LL(1) : pourquoi?
2. Donner une grammaire G' équivalente à G et qui vous semble LL(1).
3. Calculer les ensembles Premier et Suivant pour G'.
4. Donner la table d'analyse LL(1) de G'

Solution

G

- ✓ ROUTE→ INST| INST ROUTE
- ✓ INST→ go| PANNEAU TOURNE
- ✓ TOURNE→ tg| td
- ✓ PANNEAU→ ε | pan

1. Cette grammaire n'est pas LL(1) : pourquoi?

Premier(INST) = premier (INST ROUTE)

N'est pas une grammaire LL1

2. G' équivalente à G

- ✓ ROUTE → INST ROUTE'
- ✓ ROUTE' → ε | ROUTE
- ✓ INST→ go| PANNEAU TOURNE
- ✓ TOURNE→ tg| td
- ✓ PANNEAU→ ε | pan

Factorisation à gauche

Solution

G’

- ✓ ROUTE → INST ROUTE’
- ✓ ROUTE’ → ε | ROUTE
- ✓ INST→ go| PANNEAU TOURNE
- ✓ TOURNE→ tg| td
- ✓ PANNEAU→ ε | pan

Premier (PANNEAU) = **{ ε, pan }**

Premier (TOURNE) = **{tg , td}**

Premier (INST) = {go} U Premier(PANNEAU) U Premier (TOURNE si PANNEAU = ε)
= **{go , pan , tg , td }**

Premier (ROUTE) = Premier(INST) =**{go , pan , tg , td }**

Premier (ROUTE’) = { ε } U Premier (ROUTE) = **{go , pan , tg , td , ε }**

Suivant (PANNEAU) = Premier (TOURNE) = **{tg , td}**
Suivant (ROUTE) = **{ $ }**
Suivant (INST) = **{ go , pan , tg , td }** U Suivant (ROUTE si ROUTE’= ε) = **{ go , pan , tg , td , $ }**
Suivant (ROUTE’) = Suivant (ROUTE) = = **{ $ }**
Suivant (TOURNE) = Suivant (INST) = **{ go , pan , tg , td , $ }**

	First	Follow
ROUTE	{go , pan , tg , td }	{ $ }
ROUTE’	{go , pan , tg , td , ε }	{ $ }
INST	{go , pan , tg , td }	{go , pan , tg , td , $ }
TOURNE	{tg , td}	{ go , pan , tg , td , $
PANNEAU	{ ε, pan }	{tg , td}

- ✓ ROUTE → INST ROUTE’
- ✓ ROUTE’ → ε | ROUTE
- ✓ INST→ go| PANNEAU TOURNE
- ✓ TOURNE→ tg| td
- ✓ PANNEAU→ ε | pan

❑ Table d’analyse

	tg	**td**	**go**	**pan**	**$**
ROUTE	ROUTE → INST ROUTE’	ROUTE → INST ROUTE’	ROUTE → INST ROUTE’	ROUTE → INST ROUTE’	
ROUTE’	ROUTE’ → ROUTE	ROUTE’ → ROUTE	ROUTE’ → ROUTE	ROUTE’ → ROUTE	ROUTE’ → ε
INST	INST→ PANNEAU TOURNE	INST→ PANNEAU TOURNE	INST→ go	INST→ PANNEAU TOURNE	
TOURNE	TOURNE→ tg	TOURNE→ td			
PANNEAU	PANNEAU→ ε	PANNEAU→ ε		PANNEAU→ pan	

Exercice 2

- Soit la grammaire G suivante:
  - ✓E → E ou T | T
  - ✓T → T et F | F
  - ✓F → non F | (E) | vrai | faux

1. La grammaire est-elle LL(1) ?
2. Supprimer la récursivité gauche.
3. Calculer les ensembles First et Follow des symboles variables de la nouvelle grammaire.
4. Donner la table d'analyse LL(1) de la nouvelle grammaire.
5. Donner la pile d'analyse du mot "vrai et (faux ou vrai)", et en déduire l'arbre de dérivation pour ce mot

Solution

- ✓E → E ou T | T
- ✓T → T et F | F
- ✓F → non F | (E) | vrai | faux

- ✓E → TE'
- ✓E' → ou TE' | ε
- ✓T → FT'
- ✓T' → et FT' | ε
- ✓F → non F | (E) | vrai | faux

Non récursive a gauche

Solution

- ✓E → TE’
- ✓E’ → ou TE’ | ε
- ✓T → FT’
- ✓T’ → et FT’ | ε
- ✓F → non F | (E) | vrai | faux

First(E) = First(T) =First(F)= **{ non , (, vrai , faux }**

First(E’) = **{ ou , ε}**

First(T’) = **{et , ε }**

Solution

- ✓E → TE’
- ✓E’ → ou TE’ | ε
- ✓T → FT’
- ✓T’ → et FT’ | ε
- ✓F → non F | (E) | vrai | faux

Follow (E) = {) , $ }

Follow (E’) = {) , $ }

Follow (T) = { ou,) , $ }

Follow (T’) = { ou,) , $ }

Follow (F) = { et , ou,) , $ }

Solution

Table d'analyse

	et	ou	Non	vrai	faux	(	)	$
E			E→TE'	E→TE'	E→TE'	E→TE'		
E'		E'→ouTE'					E'→ε	E'→ε
T			T→FT'	T→FT'	T→FT'	T→FT'		
T'	T'→etFT'	T'→ε					T'→ε	T'→ε
F			F→nonF	F→vrai	F→faux	F→(E)		

Vérification de mot : vrai et (faux ou vrai) (sur tableau)

Analyse Syntaxique Ascendante
LR

Analyse syntaxique Ascendante

LR : Left to right – Rightmost derivation

- Cette classe de méthodes **ascendantes** couvre la méthode d'analyse déterministe la plus générale connue applicable aux grammaires non ambiguës.

- Elle présente les avantages suivants :
  - Détection des erreurs de syntaxe le plus tôt possible, en lisant les **terminaux** de gauche à droite.
  - Analyse de toutes les constructions syntaxiques des langages courants.

Analyse syntaxique Ascendante

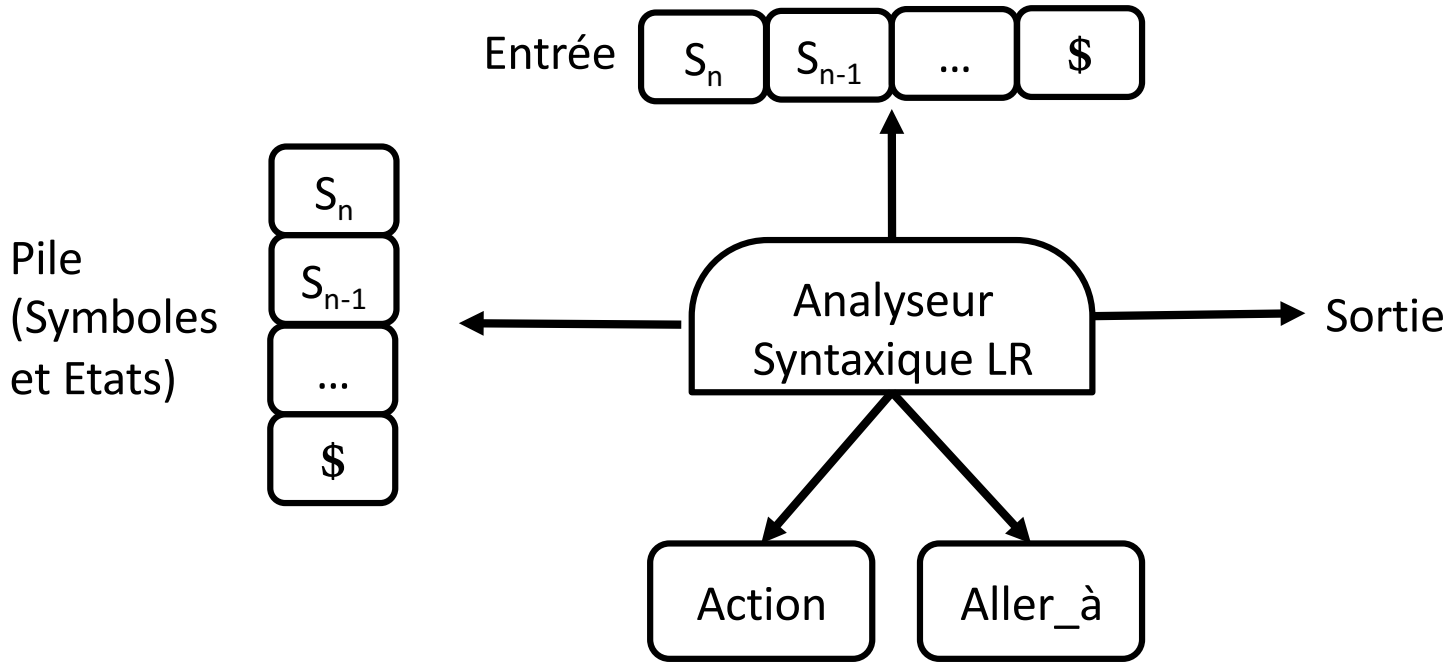
LR : Left to right – Rightmost derivation

- C'est la méthode la plus générale d'analyse syntaxique par décalage-réduction sans retour-arrière.
- Nous pouvons construire des analyseurs **LR** reconnaissant quasiment toutes les constructions des langages.
- La classe des grammaires analysées est un sur-ensemble de la classe des grammaires analysées en **LL**.

Analyse syntaxique Ascendante

LR : Left to right – Rightmost derivation

Architecture générale



Analyse syntaxique Ascendante
LR

- Les méthodes **LR** sont les plus générales, au prix de tables d'analyse volumineuses.
- La méthode **LR** comprend plusieurs cas particuliers, correspondant au même algorithme d'analyse :
  - **SLR** où **S** signifie **simple** : c'est la construction de l'automate **LR** à partir de la grammaire. Transitions données uniquement par la table ALLER_A.
  - **LALR** où **LA** signifie LookAhead **(YACC/Bison)** : ce cas couvre beaucoup de langages, avec une taille de table d'analyse de la même taille que **SLR**.
- Les méthodes **LR** construisent l'**arbre d'analyse** de dérivation en ordre inverse, en partant des feuilles.

Analyse syntaxique Ascendante
SLR

- Une position d'analyse **LR** placé dans le corps de chaque production de la grammaire est schématisée par un point •.

- Ce • indique que nous avons accepté ce qui précède dans la production, et que nous sommes prêts à accepter ce qui suit le point.

Analyse syntaxique Ascendante

LR : Left to right – Rightmost derivation

Exemple

expression ➔ expression • ’’ + ‘’ terme

- L’idée centrale de la méthode **LR** est : Étant donnée une position d’analyse •, nous cherchons à obtenir par **fermeture transitive** toutes les possibilités de continuer l’analyse du texte source, en tenant compte de toutes les productions de la grammaire par décalage ou réduction.

Analyse syntaxique Ascendante

SLR

Décalage – Réduction (shift-reduce)

- Réduction (REDUCE) : remonter la dérivation du handle en chaînage arrière

- Décalage (SHIFT) : Quand la frontière haute ne contient aucun « manche », l'analyseur repousse (décale) la frontière en ajoutant un token à droite de la frontière.

Analyse syntaxique Ascendante

SLR

Décalage – Réduction (shift-reduce)

Exemple

Pile

Flux d'entré

Au départ

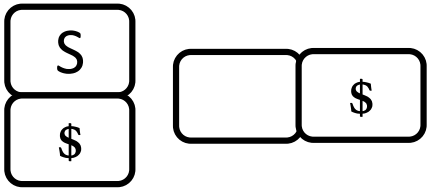
En cours

- On décale de l'entrée vers la pile 0, 1 ou plusieurs symboles jusqu'à ce qu'un manche se trouve en sommet de pile.
- On le remplace par la partie gauche de la production

Fin

- Détection d'une Erreur

ou



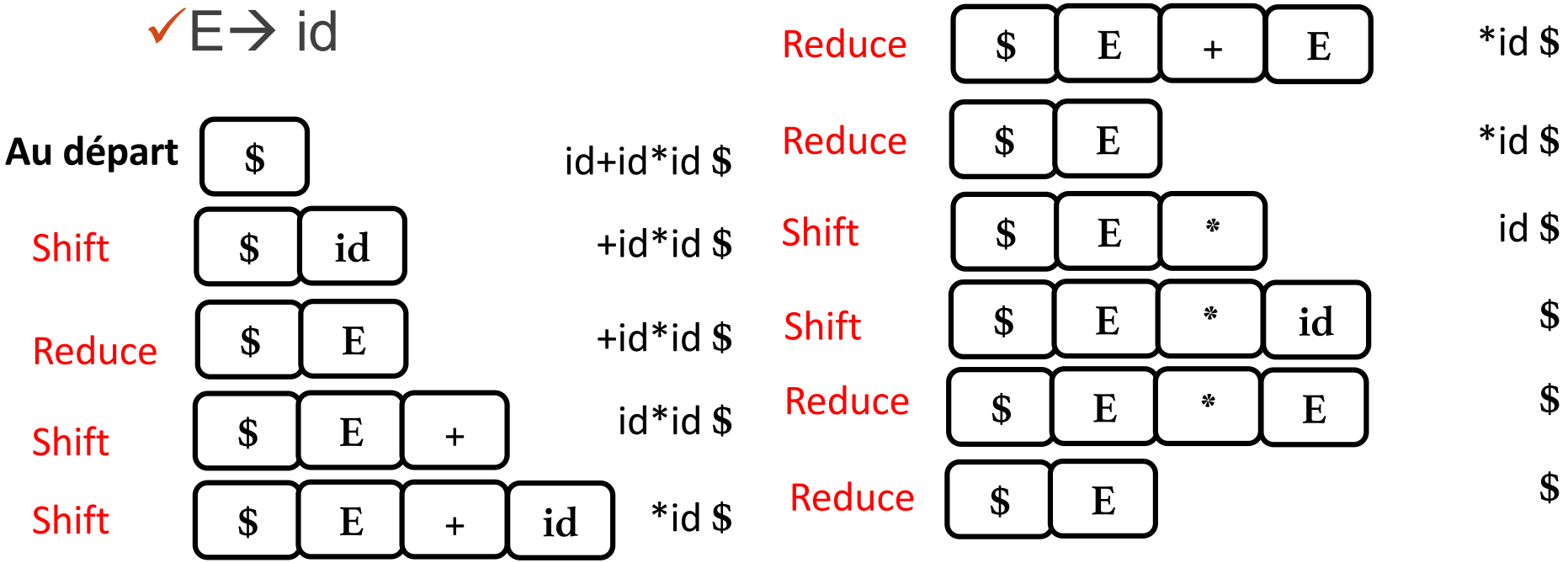
Analyse syntaxique Ascendante

Analyseur LR

Décalage – Réduction (shift-reduce)
Exemple

- ✓E → E+E
- ✓E →E * E
- ✓E → (E)
- ✓E→ id

Flux d'entrée



Analyse syntaxique Ascendante

Fermeture transitive

Définition

- **Transitive** signifie que nous propageons la connaissance que nous avons de la position d'analyse en tenant compte des productions définissant la notion non terminale que nous sommes prêts à accepter.
- **Fermeture** signifie que nous faisons cette propagation de toutes les manières combinatoires possibles, jusqu'à saturation.

Analyse syntaxique Ascendante

LR : Left to right – Rightmost derivation

Exemple

- Une position d'analyse est de la forme:
  - notion ➔ préfixe • non-terminal suffixe
- Sa **fermeture transitive** (transitive closure) se construit suivant toutes les productions définissant la notion **non-terminal** de la forme:
  - **Non-terminal** ➔ corps
- Nous ajoutons à l'état d'analyse le point • pour marquer son début :
  - **Non-terminal** ➔ • corps

Analyse syntaxique Ascendante

Analyse SLR

Exemple

- Soit la grammaire G, avec des productions récursives à gauche suivantes :
  - ✓S→ exp
  - ✓exp →exp + term | term
  - ✓term → term * fact | fact
  - ✓fact → (exp) | Entier

- L'**axiome** S permet d'avoir qu'un seul état accepteur (méthode ascendante, Grammaire Augmentée).

Analyse syntaxique Ascendante

SLR

Exemple

- Au début de l'analyse nous nous trouvons dans la position initiale : noté **Etat_0** dans l'automate **SLR**.
  - ✓S→ • exp
- Nous n'avons encore rien consommé et nous sommes prêt à accepter une exp :
  - ✓exp → • exp + term
  - ✓exp → • term
- D'après les productions de notre grammaire nous sommes dans l'une des positions d'analyse initiales suivantes:
  - ✓term → • term * fact
  - ✓term → • fact
  - ✓fact → • (exp)
  - ✓fact → • Entier

Analyse syntaxique Ascendante

Analyse SLR

Exemple

- De **l'état_0** initial, nous pouvons accepter tout ce qui se trouve à droite du point d'analyse; nous nous retrouvons alors dans un des états suivants :

Etat_1 // accepter exp

S→ exp •

exp → exp • + term

Etat_2 // accepter term

exp → term •

term → term • * fact

Etat_3 // accepter fact

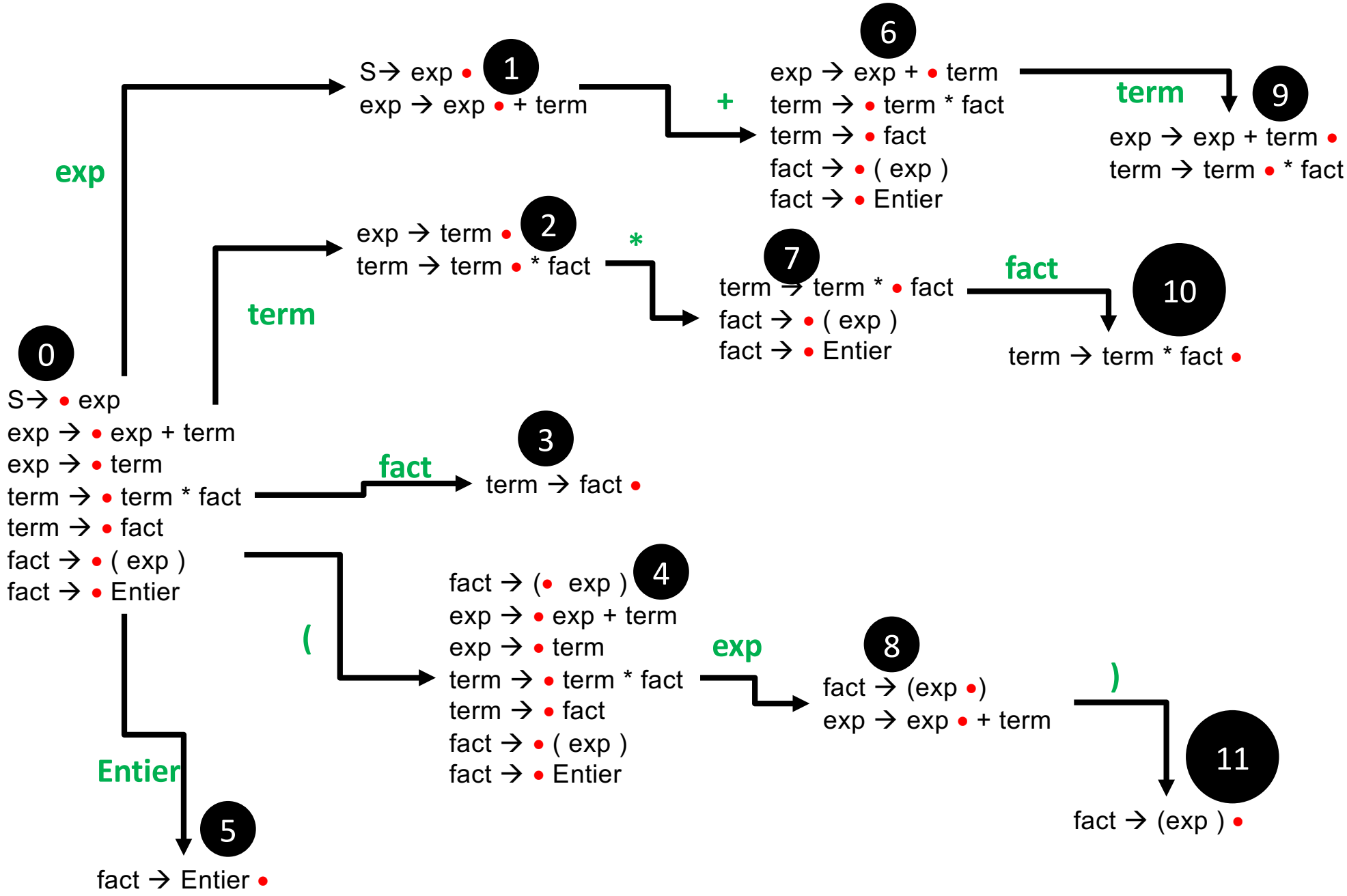
term → fact •

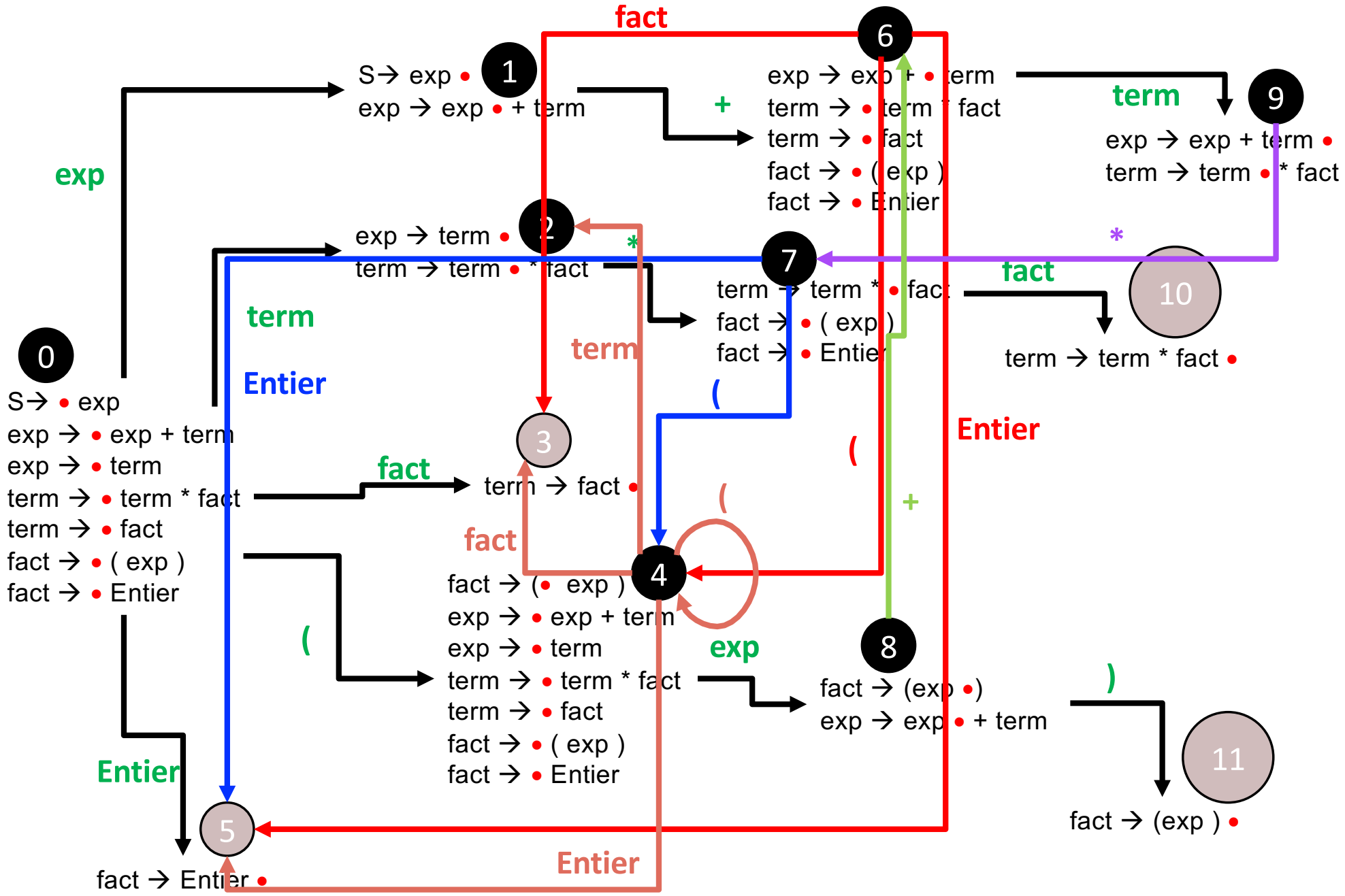
Etat_4 // accepter (

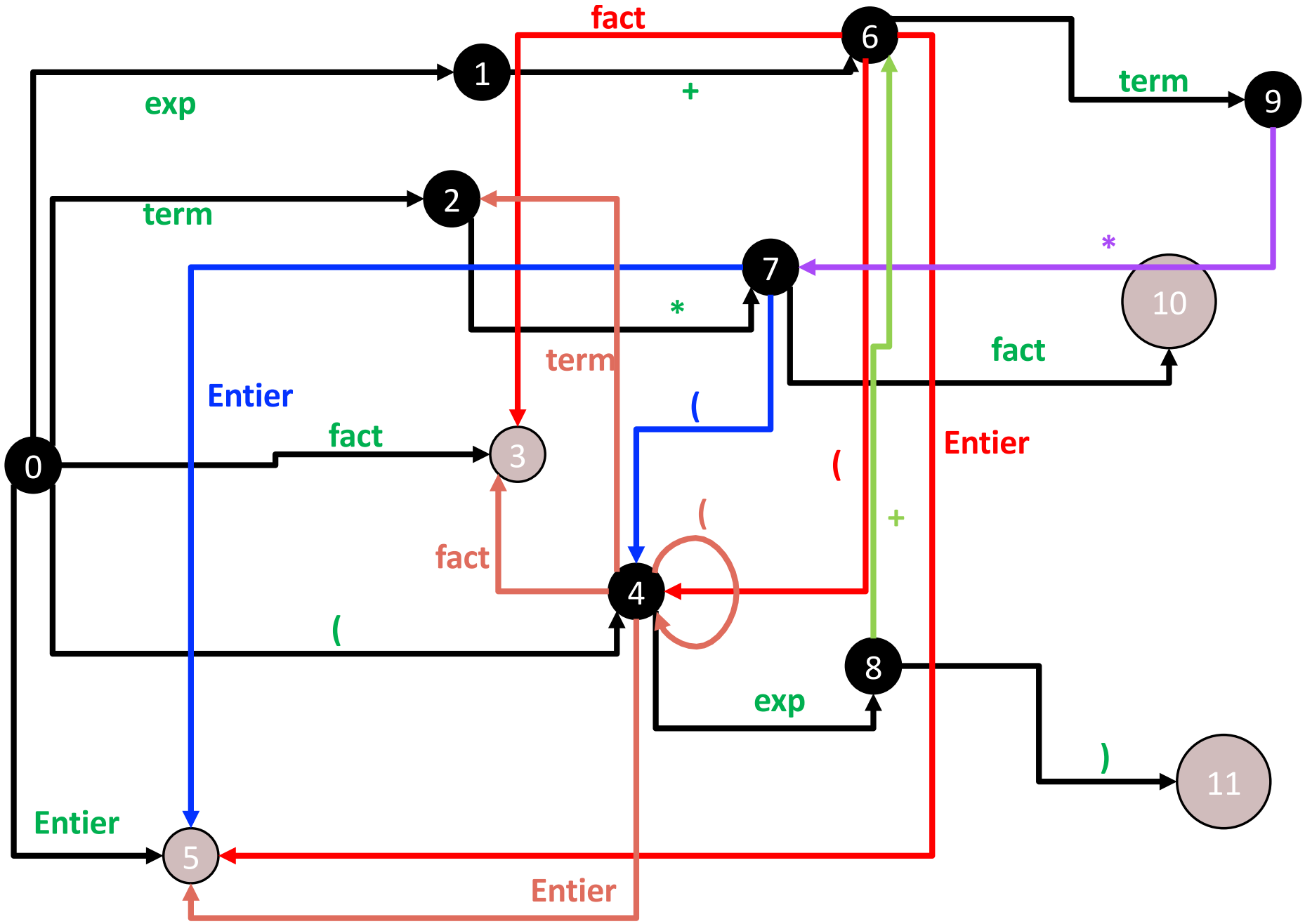
term → (• exp)

Etat_5 // accepter Entier

term → Entier •


0
S→ • exp
exp → • exp + term
exp → • term
term → • term * fact
term → • fact
fact → • (exp)
fact → • Entier
exp
term
fact
(
Entier
1
S→ exp •
exp → exp • + term
2
exp → term •
term → term • * fact
3
term → fact •
4
fact → (• exp)
exp → • exp + term
exp → • term
term → • term * fact
term → • fact
fact → • (exp)
fact → • Entier
5
fact → Entier •
6
exp → exp + • term
term → • term * fact
term → • fact
fact → • (exp)
fact → • Entier
+
7
term → term * • fact
fact → • (exp)
fact → • Entier
*
8
fact → (exp •)
exp → exp • + term
exp
9
exp → exp + term •
term → term • * fact
term
10
term → term * fact •
fact
11
fact → (exp) •
)


0
S→ • exp
exp → • exp + term
exp → • term
term → • term * fact
term → • fact
fact → • (exp)
fact → • Entier
1
S→ exp •
exp → exp • + term
2
exp → term •
term → term • * fact
3
term → fact •
4
fact → (• exp)
exp → • exp + term
exp → • term
term → • term * fact
term → • fact
fact → • (exp)
fact → • Entier
5
fact → Entier •
6
exp → exp + • term
term → • term * fact
term → • fact
fact → • (exp)
fact → • Entier
7
term → term * • fact
fact → • (exp)
fact → • Entier
8
fact → (exp •)
exp → exp • + term
9
exp → exp + term •
term → term • * fact
10
term → term * fact •
11
fact → (exp) •
exp
term
Entier
fact
(
+
*
)



Automate final

Analyse syntaxique Ascendante

SLR

Construction de la table d'analyse: Action et Aller_à

- Si l'état **état_i** contient une position d'analyse de la forme: notion ⇒ préfixe ● **terminal** suffixe
- et que: Aller_à(état_i, terminal) = état_destination
  - alors on choisit:
  - **ACTION** (état_i, terminal) = **SHIFT** état_destination
- Si l'état état_i contient une position d'analyse: notion ⇒ **corps** ● où notion n'est pas S,
  - alors pour tout terminal de **Suivant(notion)** on fixe:
  - **ACTION**(état_i, terminal) = **REDUCE** notion ⇒ corps
- Si l'état **état_i** contient la position d'analyse : S ⇒ **axiome** ●
  - alors on choisit :
  - **ACTION**(état_i, **$**) = **ACCEPT (dernier REDUCE)**
- Toutes les entrées de la table **action** qui n'ont **pas été garnies** par les quatre considérations ci-dessus sont marquées par:
  - **ACTION**(état_i, terminal_i) = **ERROR**
- on **garde** le contenu de la table **ALLER_A** pour toutes les entrées dont le second argument est une notion **non terminale**

Analyse syntaxique Ascendante

Analyse SLR

Construction de la table Action

Etat	Entier	+	*	(	)	$
0	S 5			S 4		
1		S 6				Acc
2		R 2	S 7		R 2	R 2
3		R 4	R 4		R 4	R 4
4	S 5			S 4		
5		R 6	R 6		R 6	R 6
6	S 5			S 4		
7	S 5			S 4		
8		S 6			S 11	
9		R 1	S 7		R 1	R 1
10		R3	R 3		R 3	R 3
11		R5	R5		R 5	R 5

Etats x $\sum$ = T-->états

S i= Shift, décaler
et empiler l'état i

R j = Reduce avec la régle j

R1: exp → exp + term
R2: exp → term
R3: term → term * fact
R4: term → fact
R5: fact → (exp)
R6: fact → Entier

Analyse syntaxique Ascendante

Construction de la table Aller_à

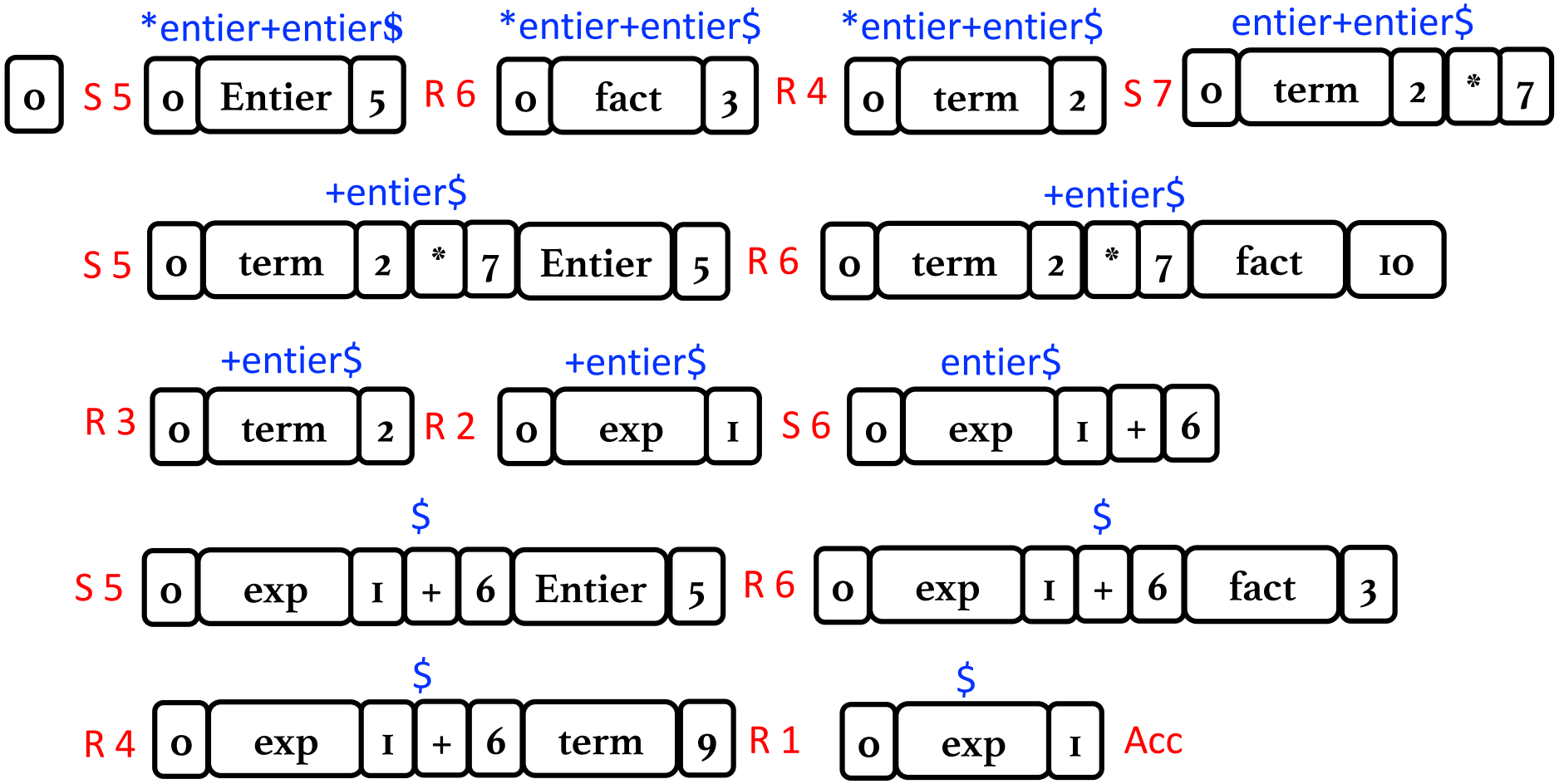
Etats x $\sum$ = NT-->états

Etat	exp	term	fact
0	1	2	3
1			
2			
3			
4	8	2	3
5			
6		9	3
7			10
8			
9			
10			
11			

Exemple

R1: exp → exp + term
R2: exp → term
R3: term → term * fact
R4: term → fact
R5: fact → (exp)
R6: fact → Entier

Flux d'entrée



Analyse sémantique

Analyse Sémantique

- Contrôle des types
  - Affectation d'une variable d'un certain type avec une expression de type diffèrent
  - Référence à une variable inappropriés :
    - Tab[I] : erreur si Tab n'est pas un tableau
    - X + I : erreur si X n'est pas un entier ou un réel
    - i<= 10 : erreur si i n'est pas une variable numérique

Analyse Sémantique

- Contrôle d'existence et d'unicité
  - Un identificateur doit être déclaré avant d'être utilisé
  - Ne pas déclarer 2 fois le même identificateur dans le même contexte
  - Gérer la portée des identificateurs
- Contrôle du flux d'exécution
  - On peut re-déclarer en local un identificateur déjà déclaré en global
  - L'instruction doit se trouver à l'intérieur d'une fonction et doit concerner une donnée de type attendu

Analyse Sémantique

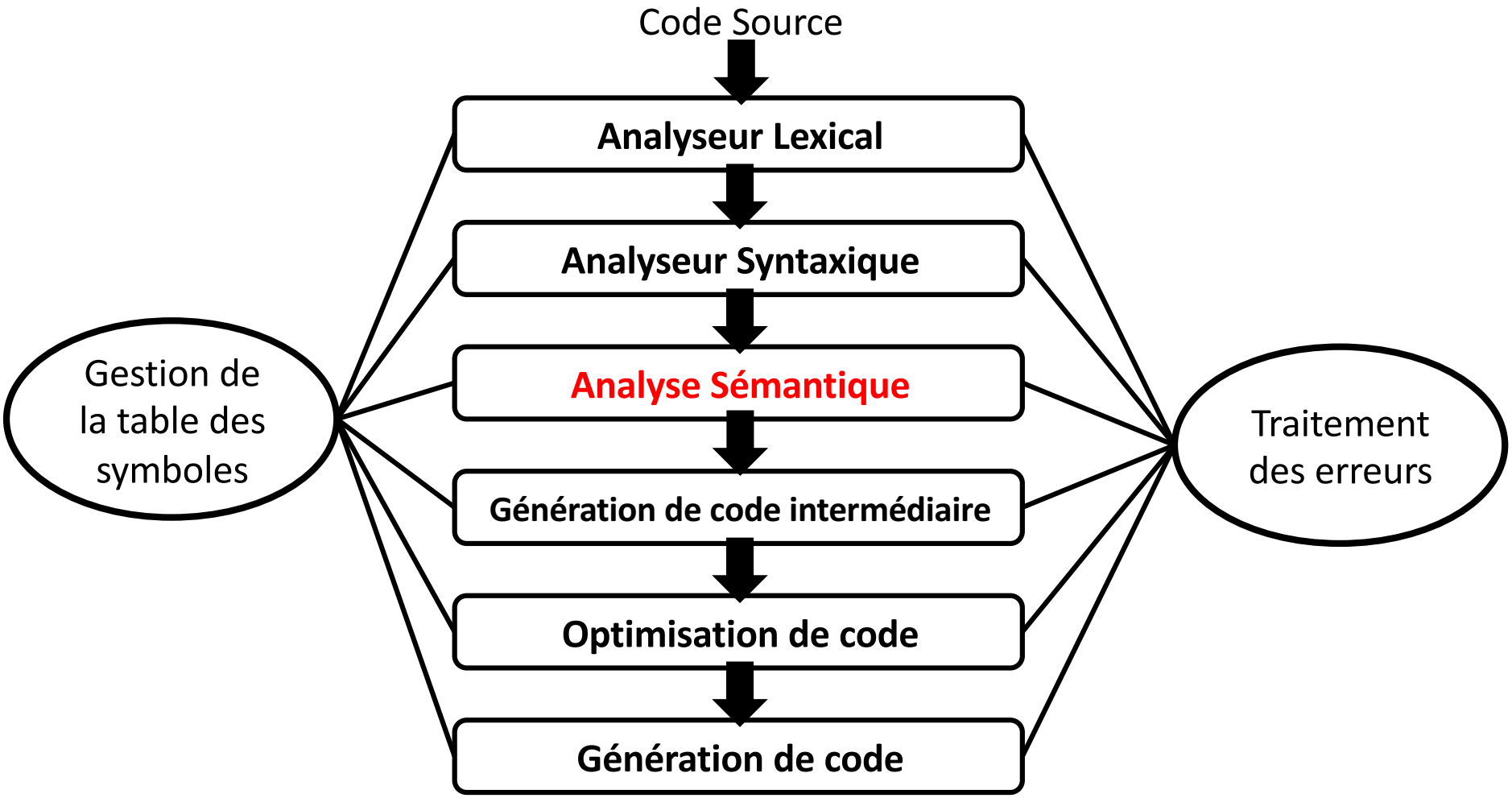
- A chaque symbole de la grammaire: on associe un ensemble d'attributs
- Chaque symbole (terminal ou non) possède un ensemble d'attributs (des variables)
- A chaque production: un ensemble de *règles sémantiques* pour calculer la valeur des attributs associés.
- Chaque règle possède un ensemble de règles sémantiques
  - Une règle sémantique est une suite d'instructions algorithmiques
- La grammaire et l'ensemble des règles sémantiques constituent la traduction dirigée par la syntaxe.

Analyse Sémantique

Analyseur sémantique :

- parcourt l'arbre syntaxique
- sauvegarde des informations à propos
  - Des types des variables (selon leur déclaration)
  - Des dimensions des variables (selon leur déclaration)
  - Taille de stockage (selon le type et la dimension),
- Utilise ces informations pour vérifier la correction des types de différentes expressions et instructions du code
- Détermine où il doit insérer une conversion entre des types de données (e.g. transformer un flottant en un entier)

Analyse sémantique



Analyse Sémantique

Grammaire attribuée

- Une **grammaire attribuée** est constituée
  - d'une grammaire hors-contexte
  - Augmentée
    - d'un ensemble d'attributs
    - d'un ensemble de règles qui spécifient certains calculs (actions sémantiques)
    - Chaque règle définit la valeur d'un attribut (variable), fonction des valeurs des autres attributs
    - Chaque règle associe l'attribut (qu'elle décrit) à un symbole de la grammaire
    - Chaque instance des symboles de la grammaire apparaissant dans l'arbre syntaxique correspondent à une instance de l'attribut
    - d'un ensemble de conditions (prédicats)

Analyse Sémantique

Grammaire attribuée

- Vérification de type (analyse contextuelle : *context-sensitive analysis*)
- Construction de représentation intermédiaires (*AST*)
- Calcul dirigé par la syntaxe (*interpréteur*)
- Génération de code (*compilateur*)

Analyse Sémantique

Attribut: Notation

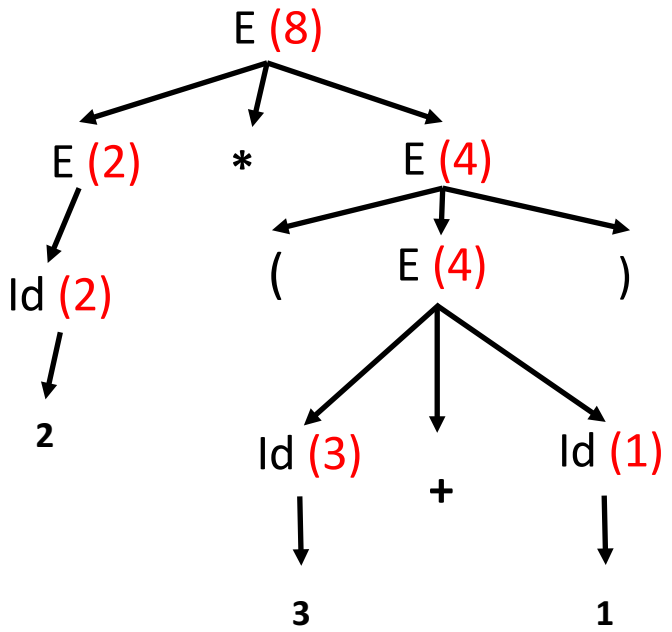
- On notera X.a l'attribut a du symbole X.
- S'il y a plusieurs symboles X dans une production :
  - on notera $X^{(0)}$ s'il est en partie gauche
  - $X^{(1)}$ si c'est le plus à gauche de la partie droite,
  - ..., $X^{(n)}$ si c'est le plus à droite de la partie droite
- On appelle *arbre syntaxique décoré* un arbre syntaxique sur les nœuds duquel on rajoute la valeur de chaque attribut

Analyse Sémantique

Arbre décoré: Exemple

- ✓E → E+E
- ✓E →E * E
- ✓E → (E)
- ✓E→ id
- ✓id→0 | 1 |…. | 9

E =2*(3+1)



Analyse Sémantique

Attributs synthétisés

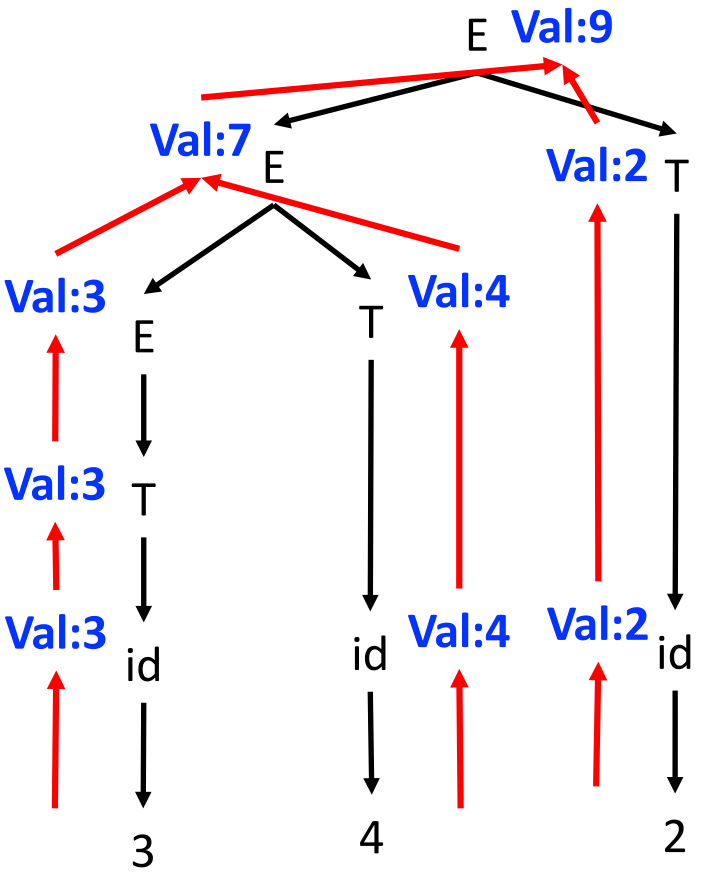
- Un attribut est *synthétisé* si sa valeur, à un nœud d'un arbre syntaxique, est déterminée à partir de valeurs d'attributs des fils de ce nœud.
- L'attribut de la partie gauche est calculé en fonction des attributs de la partie droite.
- Le calcul des attributs se fait des feuilles vers la racine.
- Les attributs synthétises peuvent être donc évalués au cours d'un parcours ascendant de l'arbre de dérivation

Analyse Sémantique

Attributs synthétisés : Exemple

- ✓E → E+T
- ✓E →T
- ✓T → (E)
- ✓T→ id

- ✓$E^{(0)}$.val → $E^{(1)}$.val + T.val
- ✓E.val →T.val
- ✓T.val → (E.val)
- ✓E.val→ id.val



Analyse Sémantique

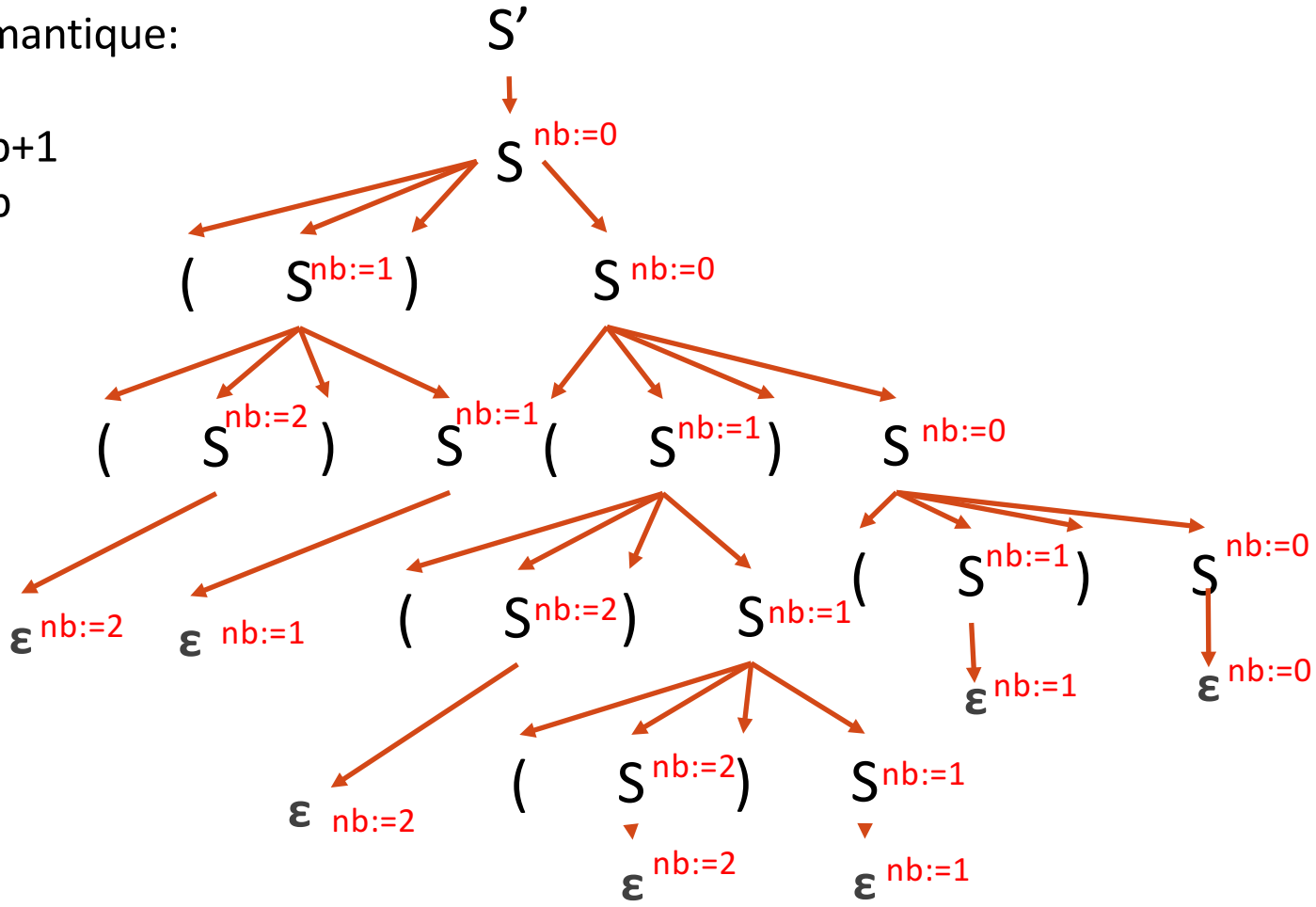
Attributs hérités

- Un attribut *hérité* est un attribut dont la valeur à un nœud d'un arbre syntaxique est définie en terme des attributs du père et/ou des frères de ce nœud
  - Le calcul est effectué à partir du non terminal de la partie gauche et éventuellement d'autres non terminaux de la partie droite
- Si les attributs d'un nœud donné ne dépendent pas des attributs de ses frères droits, alors les attributs hérités peuvent être facilement évalués lors d'une analyse descendante

Analyse Sémantique

Attributs hérités : Exemple

Régles.	Action sémantique:
S'→S	S.nb:=0
S→(S)S	$S^{(1)}$:=$S^{(0)}$.nb+1
	$S^{(2)}$:=$S^{(0)}$.nb
S→ ε	Ecrire S.nb



Arbre décoré pour le mot:

(( )) (( ) ( )) ()

Analyse Sémantique

Exemple

- Soit les règles suivantes avec les actions sémantiques

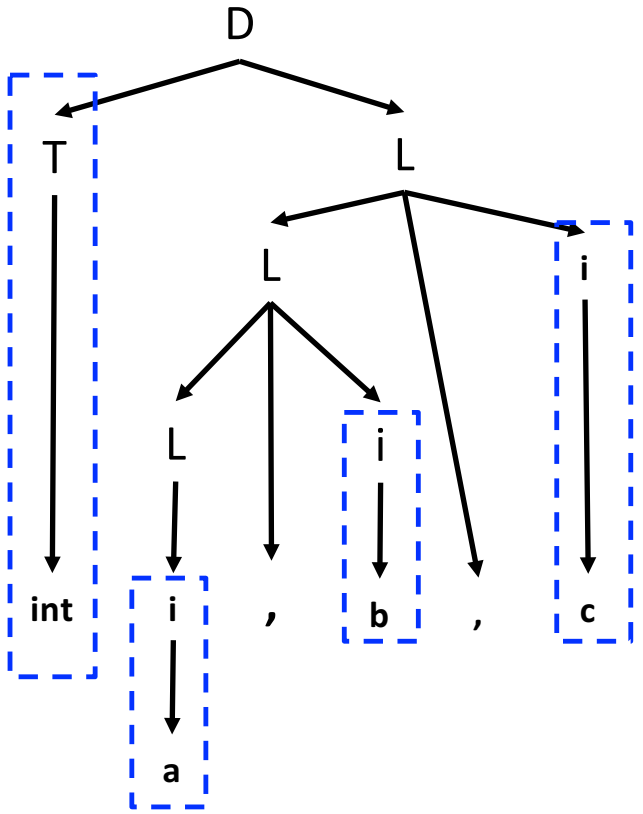
Grammaire	Règles sémantique
D→T L	{ D.type = T .type L.type = T .type}
T→int	**{ T.type = int }**
T→real	{ T.type = real }
L→L,i	{ $L^{(1)}$.Type = $L^{(0)}$.Type }
	{ addSymbTable(i.token,$L^{(1)}$.type) }
L→i	{ i.Type = $L^{(0)}$.Type }
	{ addSymbTable(i.token , L.type) }

D représente le début de la déclaration
i représente les identificateurs
T représente le type des variable
L la liste des variables séparées par une virgule

Exemple : Analyse syntaxique

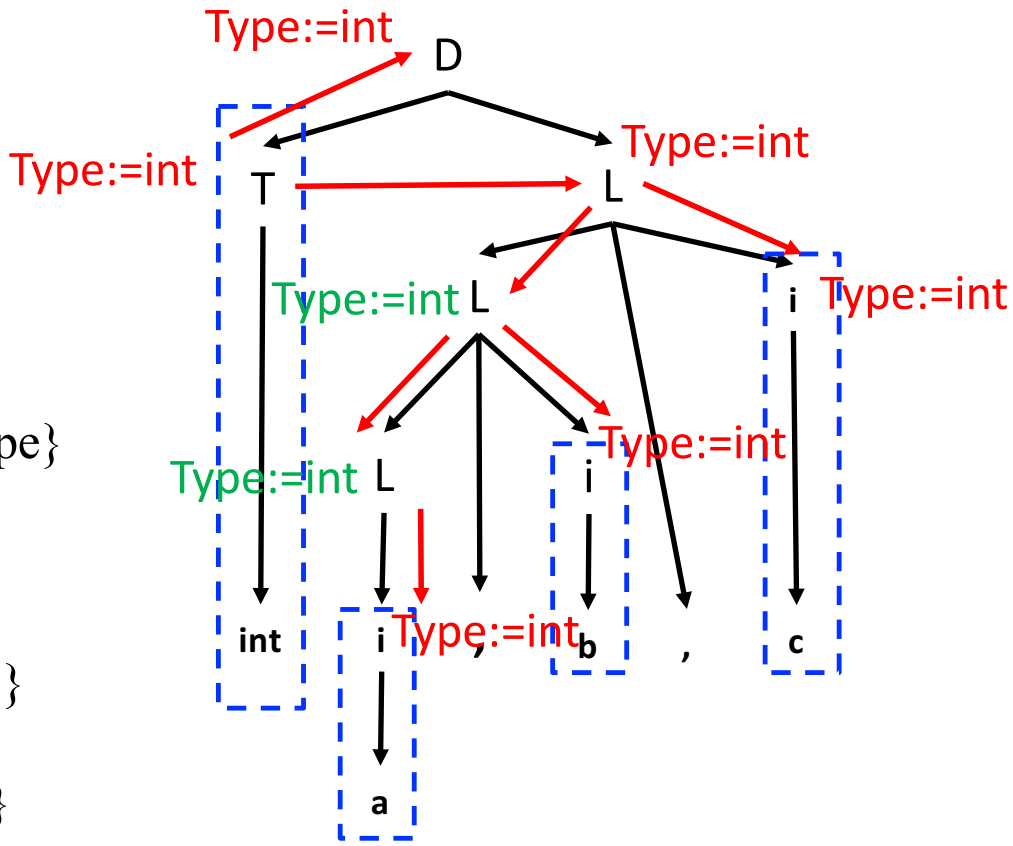
Soit la déclaration suivante : **int a,b,c**

D→T L
T→int
T→real
L→L,i
L→i



Analyse Sémantique

int a,b,c



D→T L { D.type = T .type L.type = T .type}
T→int { T.type = int }
T→real { T.type = real }
L→L,i { $L^{(1)}$.Type = $L^{(0)}$.Type }
{ addSymbTable(i.token,$L^{(1)}$.type) }
L→i { i.Type = $L^{(0)}$.Type }
{ addSymbTable(i.token , L.type) }

Analyse Sémantique

- Grammaires **s-attribuées**: possèdent seulement des attributs **synthétisés**. Elles sont donc compatibles avec une évaluation effectuée en parallèle avec une analyse syntaxique ascendante
  - Les générateurs d'analyseurs ascendants (LR) ne permettent d'utiliser que des grammaires s-attribuées
- Grammaires **l-attribuées**: possèdent seulement des attributs **hérités**. Les attributs d'un nœud ne dépendent que du père ou des frères a sa gauche, d'où leur compatibilité avec une évaluation effectuée en parallèle avec une analyse syntaxique descendante

Générateur d'Analyseur Syntaxique
YACC/BISON

Générateur d'Analyseur Syntaxique
YACC/BISON

- **YACC** synthétise une grammaire (**LALR(1) de la famille des LR**) et produit le texte source d'un analyseur syntaxique du langage engendré par la grammaire donnée.

- Le langage cible c'est celui dans lequel le code de l'analyseur syntaxique synthétisé est écrit, en général C, C++, Java, et d'autres.

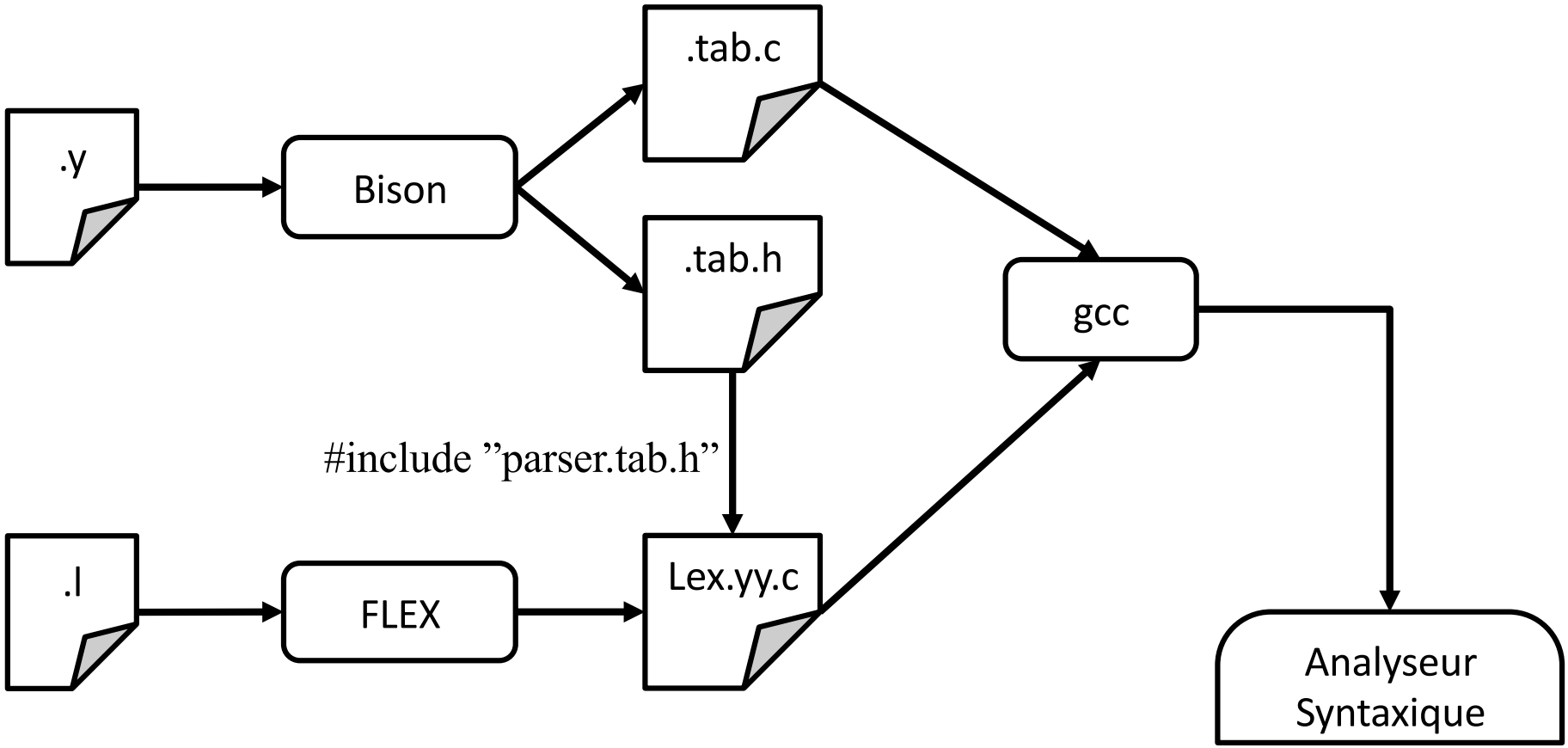
Générateur d'Analyseur Syntaxique
YACC/BISON

- YACC: générateur d'analyseur syntaxique.

- Prend en entrée la définition d'un schéma de traduction (grammaire + actions sémantiques). Produit un analyseur syntaxique pour le schéma de traduction.

- Il existe plusieurs versions de YACC, nous utiliserons ici bison.

Générateur d'Analyseur Syntaxique
YACC/BISON
Schéma



Générateur d'Analyseur Syntaxique
YACC/BISON
Format fichier .y

```
%{

/* Déclarations */

%}

/* Définitions Bison */

%%

/* Règles de production */

%%

/* Code utilisateur */
```

Générateur d'Analyseur Syntaxique
YACC/BISON
Format fichier .y

- Les déclarations globales comprend :
  - L'inclusion des fichiers d'en-tête.
  - La déclaration des variables globales.
  - La déclaration des fonctions globales.

```
%{
int variable1 = 0;
int variable2 = 0;
%}
```

Générateur d'Analyseur Syntaxique
YACC/BISON
Format fichier .y

- Les déclarations de Bison comprend :
  - La définition des noms des symboles terminaux et non-terminaux.
  - La spécification de la précédence des opérateurs.
  - La définition des types de données des valeurs sémantiques des divers symboles.

%token terminal1
%token terminal2

Générateur d'Analyseur Syntaxique
YACC/BISON
Format fichier .y

- Les règles de la grammaire définissent
  - Comment construire chaque non-terminal à partir de ses composants.
  - La fonctions auxiliaires sont des fonctions écrites en langage C ou (C + +).

```
Non-terminal : corps_1 {prod_ou_action_1}
             | corps_2 {prod_ou_action_2}
             | corps_3 {prod_ou_action_3}
             |  …
             | corps_n {prod_ou_action_n}
             ;
```

- L'analyse lexicale produit une séquence de **terminaux**, et l'analyse syntaxique accepte ou rejette cette séquence.

Générateur d'Analyseur Syntaxique
YACC/BISON
Format fichier .y

- **Un peu de sémantique pour manipuler les valeurs.**
- **Attributs** : À chaque symbole est associée une valeur, qui est de type int par défaut. Cette valeur peut être utilisée dans les actions sémantiques (comme un attribut synthétisé).
  - Le symbole **$$** référence la valeur de l'attribut associé au **non-terminal** de la partie gauche.
  - Le symbole **$i** référence la valeur associée au i-ème symbole (**terminal** ou **non-terminal**) ou action sémantique de la partie droite.
  - Lorsque aucune action n'est pas indiquée, **YACC** génère par défaut l'action : **{$$ = $1; }**

Générateur d'Analyseur Syntaxique
YACC/BISON
Exemple

- Soit la grammaire suivante :
  - expr ➔ term
  - | term + expr
  - | term - expr

```
expr : term                    { printf( ''expression'');}
     | term + expr             { printf( ''+'');}
     | term - expr             { printf( '' -'');}
     ;
```

Générateur d'Analyseur Syntaxique
YACC/BISON
Exemple

```
expr : term                    { printf( ''expression'');}
     | term + expr             {printf( ''+'');}
     | term - expr             {printf( '' -'');}
     ;
```

```
expr : term                    {$$ =$1; //par defaut}
     | term + expr             {$$ =$1 + $3;}
     | term - expr             {$$ =$1 - $3;}
     ;
```

Exemple

Fichier Lex

```
%{
#include "global.h"
#include "calcularice.tab.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);
```

```
bison -d calcularice.y
flex calcularice.l
gcc -c lex.yy.c -o calcularice.lex.o
gcc -c calcularice.tab.c -o calcularice.y.o
gcc -o calcularice calcularice.lex.o calcularice.y.o -ll -lm
```

Fichier Yacc

```
%{
#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(); }
```