Notions sur la TRADUCTION

0) Motivation et objectifs 2

1) Notions de base 9

2) Analyse lexicale 14

3) Automates finis 19

4) Grammaires 30

5) Analyse syntaxique 39

A) Utilisation Flex et Bison 48

B) Bibliographie 55

0) Motivation et Objectifs

0.1 - Des langages partout ...3
0.2 - De la théorie 5
0.3 - Et en pratique 7
0.4 - Objectifs du cours 8

0.1 - DES LANGAGES PARTOUT (1/2)

Langages de programmation	HTML, XML, JavaScript, ... C, C++, Java, C#, Ada, Ruby, Basic, … Perl, Python, PHP, Tcl/Tk, ... Smalltalk, Prolog, Lisp, Scheme, ...
Déploiement, administration, réseau, systèmes répartis	shells (sh,csh,ksh,bash …) commandes "filtres" (grep, sed, awk …) commandes de systèmes (automates, serveurs …) spécifications Make, Ant, … Xpath, routage basé sur XML, ...
Génie logiciel, transformations ...	édition syntaxique, pretty printing, ... extraction de références croisées, décompilation (byte-code, langage machine) recherche de la structure de programmes
Conception de systèmes matériels ou logiciels	langage de spécifications (B), description de protocoles (Lotos), langages de simulations (OpNet), spécifications hardware (VHDL), ...

0.1 - DES LANGAGES PARTOUT (2/2)

Documents, multimédia, description objets, scènes, animations...	Latex, ps, PDF, Flash, SVG ... XML, XUL, Xalan, Xerces … VRML, Povray, RenderMan … ActionScript, MEL ... Shaders (HLSL, GLSL, Cg …)
Recherche, extraction	recherche d'information sélective, filtrage analyse de log, business reports …) langages de requêtes DB web mining, XMLquery ...
Interaction homme / machine, comportements, jeux	description, reconnaissance de phonèmes description, reconnaissance de phrases simples aide à la traduction de langage naturel langages gestuels, description de stratégies ...
Description, reconnaissance de formes	biologie (croissance des plantes, structure des gènes …) analyse de séries temporelles (courbes, bourse …) analyse de traces (pannes, interactions d 'automates …) recherche de formes communes (workflows …)
Systèmes formels, solveurs	expressions mathématiques, descriptions géométriques Maple, Mathematica, ... résolution de problèmes logiques (Prolog, OPLStudio)

0.2 - DE LA THEORIE (1/2)

La théorie des langages ou de la calculabilité explore ce que l'on peut décrire ou calculer automatiquement

Des fondements : grammaires et systèmes formels

Hiérarchie de CHOMSKY-SCHÜTZENBERGER

	Langages/Grammaires	**Machine**
type 0	récursivement énumérables	machine de Turing
type 0b	récursifs	machine de Turing qui s'arrête
type 1	contextuels (*context-sensitive*)	automates à ressources bornées
type 2	algébriques (*context-free*)	automates à pile non déterministes
type 2b	algébriques déterministes	automates à pile déterministes
type 3	rationnels (*regular*)	automates à états finis

0.2 - DE LA THEORIE (2/2)

Machine de Turing et calculabilité ()

BANDE infinie

0	1	1	0	1	0	1	1	1	0	0	0	1	1	1	1	...

TETE Lecture, Écriture : 0 ou 1
Déplacement : Droite ou Gauche

CONTROLE un état (*« variable entière »*),
des instructions (*« table d'automate »*) :
[état courant, symbole lu, nouvel état, action]
avec action dans {0,1,D,G}

« La machine la plus simple pour les calculs les plus généraux»

-Thèse de Church-Turing : tout ce qui est calculable automatiquement est calculable avec une machine de Turing

- Il existe des fonctions non calculables ...
- L'arrêt de la machine de Turing est indécidable

A voir sur le net : « machine de Turing en Lego de l'ENS Lyon (2012)»

0.3 - ET EN PRATIQUE ?

langages type 3 = expressions régulières

- reconnues par automates à états finis

. de nombreuses commandes informatiques utilisent les e.r
. de nombreux automates matériels sont basés sur les e.r.

- ∃ outils pour reconnaître les expressions régulières en O(n)

> - ***Flex*** *(lex)* ***: un générateur d'analyseurs lexicaux***
> ***- aussi : Perl…***

langages type 2 (*ou 2bis*) = grammaires algébriques ~ "langages informatiques"

- d'un usage très fréquent, *au delà des langages de programmation*

- ∃ outils pour reconnaître une grammaire algébrique (déterministe) en O(n)

> - ***Bison*** *(yacc)* ***: un générateur d'analyseurs syntaxiques***
> ***- aussi : javacc, jaxp, ...***

yacc=yet another compiler's compiler

0.4 – OBJECTIFS DU COURS

Pouvoir utiliser les générateurs d'analyseurs Flex et Bison

- reconnaître des syntaxes de difficulté moyennes
- faire des transformations simples
- combiner analyse lexicale et syntaxique

Pouvoir utiliser efficacement les logiciels basés sur les *expressions régulières* et les *grammaires algébriques*

- shell, commandes, environnements usuels …
- plus tard selon les besoins : Xpath, analyseurs SAX et DOM, …

Pouvoir expliquer et utiliser les principales notions

- automates et expressions régulières (spécification, fonctionnement…)
- grammaires (définition, spécification, classes de grammaires…)
- principes de reconnaissance, connexion avec la complexité des algorithmes

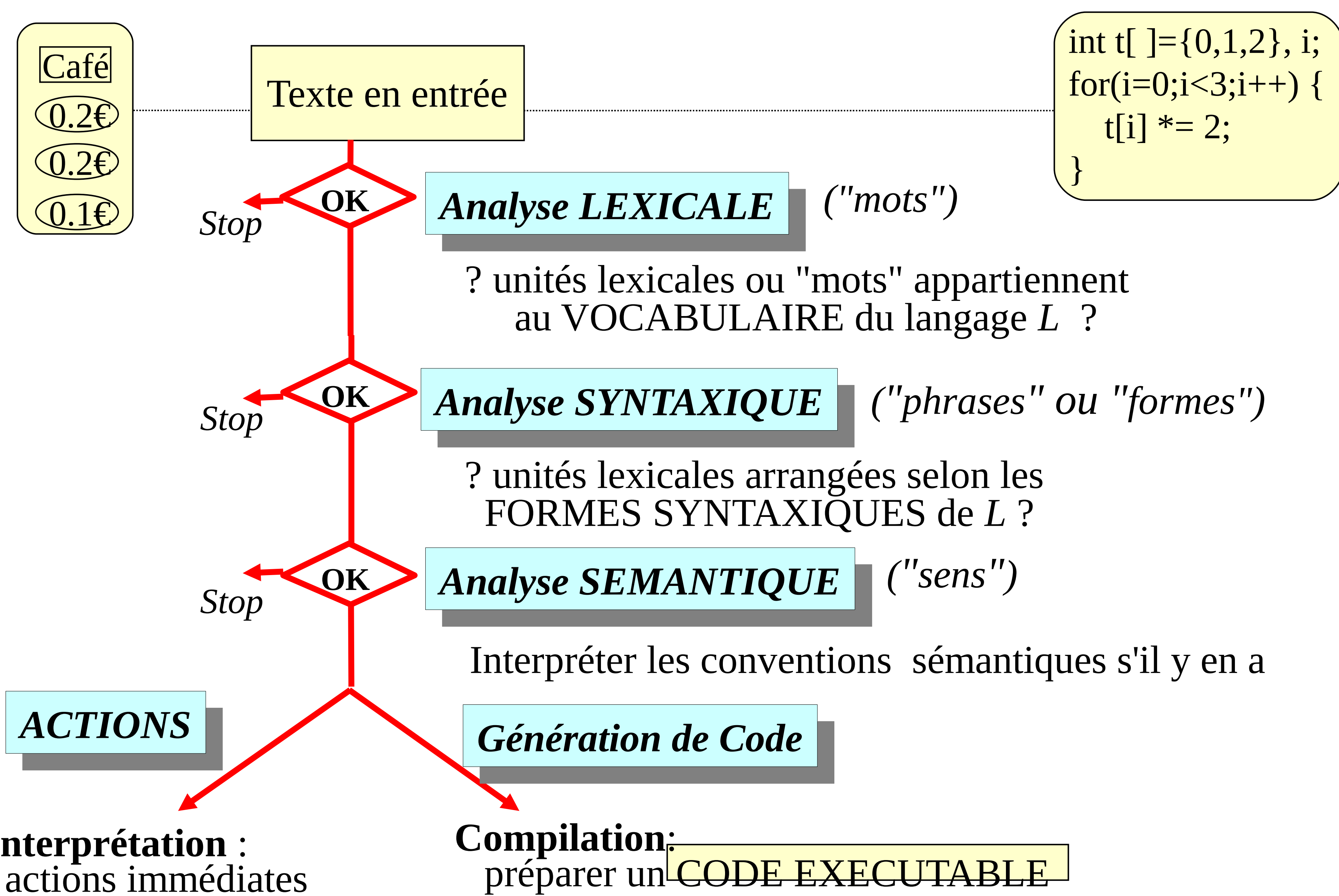
Pouvoir approfondir en cas de besoin

Notions sur la TRADUCTION

1) Notions de base

1.1 - Compilation : plusieurs phases 10
1.2 - Vocabulaire et grammaire d’un langage .. 11
1.3 - Grammaire : définition et utilisation 12
1.4 - Arbre syntaxique 13

1.1 – COMPILATION : PLUSIEURS PHASES



1.2 – VOCABULAIRE ET GRAMMAIRE

Vocabulaire ➡ **unités lexicales possibles** ("mots") **du langage cible *L***

- *un vocabulaire fini peut être défini par énumération*
- *d'autres méthodes permettent des vocabulaires **infinis** => Voir 2 - Analyse Lexicale*

Règles syntaxiques ➡ **constructions possibles pour grouper les lexèmes**

Exemple : grammaire d'un langage naturel élémentaire

Catégories lexicales   Valeurs possibles

partie lexicale

<DOT>	:=	'.'
<ART>	:=	a | an | the
<TRANS_VRB>	:=	cut | cuts
<COMM_NOUN>	:=	bill | bills
<PROP_NOUN>	:=	Bell | Verizon

- définitions des « mots » possibles dans L
= « vocabulaire terminal »
= « Tokens »

partie syntaxique

<Sentence> := *<SubjVrbCplt> <DOT>*
<SubjVrbCplt>:= *<NounGrp> <TRANS_VRB> <NounGrp>*
<NounGrp> := *<PROP_NOUN>* | *<ART> <COMM_NOUN>*

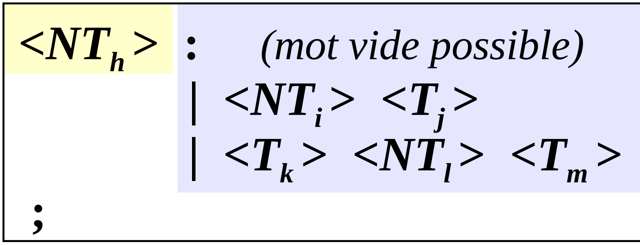
- définitions des formes possibles en fonction d'autres formes ou de catégories lexicales
- définitions éventuellement récursives

1.3 - GRAMMAIRE : DEFINITION ET UTILISATION

Grammaire G	**V_T** vocabulaire terminal	*Mots du langage*
(V_T,V_N, P, S)	**V_N** vocabulaire non-terminal	*Formes syntaxiques de la grammaire*
	P règles syntaxiques	*Règles de production de phrases*
	S axiome	*Symbole de* **V_N** *servant de départ*

Règle syntaxique pour une grammaire non contextuelle:

Membre Gauche
Un symbole de V_N



Membre Droit
Un ensemble d'alternatives (OU)
chacune est une concaténation (ET)
de symboles de V_T ou V_N

Utilisations :

a) en «<u>production</u>» de formes : Substitutions de partie gauche par partie droite

<Sentence> → *< SubjVrbCplt> <DOT>* → *<NounGrp> <TRANS_VRB> <NounGrp> <DOT>*

... → Bell cut Verizon . = *une production <u>syntaxiquement</u> correcte*

b) en «<u>reconnaissance</u>» ou «parsing» : Réductions d'une phrase en remplaçant des membres droits par les membres gauches. (réduction finale = Axiome)

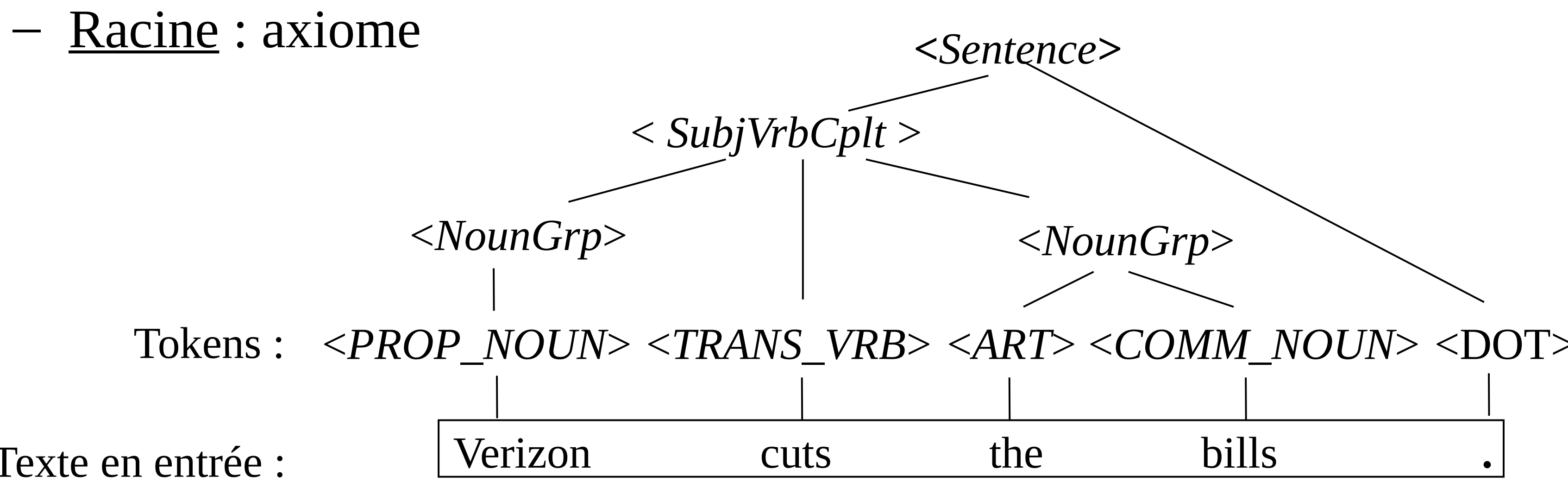
1.4 - ARBRE SYNTAXIQUE

Décrit la structure d'une forme

- reconnaissance d'une phrase conforme à la grammaire
- décrit les réductions successives de l'analyse syntaxique

Est un arbre étiqueté

- Feuille : unité lexicale reconnue, mots de V_T
- Nœud : membre gauche de règle de production, symbole de $\mathbf{V_N}$
- Fils = membres droits
- Racine : axiome



Notions sur la TRADUCTION

2) Analyse lexicale

2.1 - Introduction à l'analyse lexicale 15
2.2 - Définition du vocabulaire terminal 16
2.3 - Expressions régulières 17

2.1 - INTRODUCTION A L'ANALYSE LEXICALE

ROLE
- reconnaître les **<u>unités lexicales</u>** ou "mots"
- et leur **<u>catégorie lexicale</u>** ou "token"

PRINCIPE

- du texte en entrée :

Verizon , Bell South will cut the DSL bills .

- dérivation d'une **<u>chaîne</u>** de couples :
  - *-Identifiant de la catégorie*
  - *-Valeur dans la catégorie*

5	1	5	5	3	4	2	6	6	0
1		0	2	0	0	0	2	1	

- des structures auxiliaires

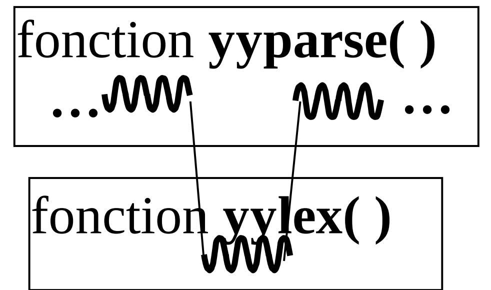
0	1	2	3	4	5	6
.	,	ART	AUX	TRANS_VERB	PROP_NOUN	COMM_NOUN
		0: the	0: will	0: cut	0: Bell	0: bill
				1: cuts	1: Verizon	1: bills
						2: DSL

NB: seule la catégorie est utile pour l'analyse syntaxique,
la valeur dans la catégorie sera reprise ensuite pour l'analyse sémantique

IMPLEMENTATION avec les outils Flex et Bison

- l'analyseur syntaxique demande quand il en a besoin la prochaine <u>unité lexicale</u>

- l'analyseur lexical retourne la <u>catégorie lexicale</u> (et passe éventuellement la valeur)



préparé avec **Bison**

préparé avec **Flex**

2.2 - DEFINITION DU VOCABULAIRE TERMINAL

PRINCIPE : *unités lexicales ou mots* : formés par concaténation de symboles élémentaires ou caractères pris dans un alphabet

DEFINITION : DIFFERENTES POSSIBILITES

1) par énumération

- vocabulaire limité, statique
- Techniques : tables, dictionnaires, énumération dans des règles de production « plates »

```
<Séparateur>  :  ‘,’ | ‘.' | ‘:’ | ‘;’   ;
<Opérateur>   :  ‘+’ | ‘-’ | ‘*’ | ‘/’   ;
```

2) par règles de production récursives

- le vocabulaire peut être infini
- Ex : syntaxe BNF (Backus-Naur Form)

```
<Nom>      :  <Lettre>
           |  <Nom >  <Lettre> ;
<Entier>   :  <Chiffre>
           |  <Entier >  <Chiffre>  ;
<Lettre>   :  ‘A’ |  ‘B’ |  …..    ;
<Chiffre>  :  ‘0’  |  ‘1’ |  …..    ;
```

3) par "Expressions Régulières"

- formalisme plus simple
- analyse automatisable
- le vocabulaire peut être infini

```
<Entier>  = [0-9]+
<Nom>  = [a-zA-Z][_a-zA-Z]*
```

2.3 - EXPRESSIONS REGULIERES (1/2)

PRESENTATION

- formalisme plus limité que les règles de production générales
- suffisant pour analyse lexicale de langages simples

UTILISATION

a) ***<u>spécification</u>*** ***d'une e.r. :***
opérateurs et méta-caractères

b) ***<u>reconnaissance</u>*** ***d'e.r. :***
par interpréteur et
construction d'un automate

Opérations fondamentales

ε	mot vide
Car.	singleton
	concaténation (implicite)
|	alternative
*	répétition N ≥ 0 fois
( )	groupement

EXEMPLE

alphabet de base : {a, b, c}

e.r. : **a + (b | c) * a ***

caractérise un ensemble infini de mots :

a
aa
ab
aba
abbb
acbcba
…

- *chaînes sur* {a,b,c}
- *commençant par au moins un* a
- *suivi(s) de* b *ou* c *en nombre quelconque*
- *éventuellement terminée par des* a

2.3 - EXPRESSIONS REGULIERES (2/2)

Répétitions

r **?**	expression *r* répétée 0 ou 1 fois (== *r* | ε)
r +	expression *r* répétée N > 0 fois (== *r r* *)
r {***k***}	expression *r* répétée exactement *k* fois

Classes de symboles : Reconnaît **un** symbole parmi un ensemble

.	Tout sauf \n
[*xyzt*]	définition par énumération de symboles
[…*a*-*z*…]	définition d’intervalle
[^…]	définition par complémentarité

Délimiteurs de contexte

^*r*	expression *r* en début de ligne
r$	expression *r* en fin de ligne

Exemple :

[A-Za-z]+	un mot alphabétique (non vide)
[aeiouy]	une voyelle
[0-9]+(\.[0-9]*)?	un nombre réel
^[\t]+$	une ligne "vide" (uniquement du blanc)

3) Automates finis

3.0 - Abrégé de théorie des langages 20
3.1 - Automates finis : exemples 22
3.2 - Automates finis : définition 24
3.3 - Exemple d'application : traitement de texte .. 25
3.4 - Reconnaissance par un automate fini 26
3.5 - Reconnaissances multiples 27
3.6 - Déterministe vs non-déterministe 28
3.7 - Langage reconnu par un automate 29

3.0 – Abrégé de THEORIE DES LANGAGES (1/2)

Définitions :

Alphabet Σ = ensemble fini de lettres

Mot w sur Σ = suite finie de lettres de Σ

Σ^* = ensemble de tous les mots = monoïde libre

Langage L = ensemble de mots $\subset \Sigma^*$

Mot vide ε, Langage vide $\emptyset \neq \{\varepsilon\}$

Opérations :

Alternative (union)

Concaténation (produit)

Répétition de Kleene (étoile) $L^* = \{\varepsilon\} \cup L \cup LL \cup LLL$ ...

Aussi intersection, complémentaire, quotient …

Questions :

$w \in L$? , $L = \emptyset$?, L fini ?, L1=L2 ?, ...

décidable, calculable, facilement calculable ?

3.0 – Abrégé THEORIE DES LANGAGES (2/2)

Langage	Rationnel	Algébrique déterministe	Algébrique	Contextuel (aussi monotone)	Récursif ou décidable	Récursivement énumérable
Machine	Automate fini	Automate à pile		Automate à ressources bornées	Turing qui s'arrête	Machine de Turing
Spécification	Expression régulière	Grammaire *context-free*		Grammaire *context-sensitive*	Grammaire générale ou *phrase-structure*	
Exemples	a^n	a^nb^n , $a^nb^nc^pd^p$, Dyck ([]())	Palindrome $w\overline{w}$ $a^nb^nc^p \cup a^pb^nc^n$ a^p et p carré	ww, $a^nb^nc^n$, $a^nb^pc^nd^p$, a^p et p premier	diag(CSL) RegExp==? Ackermann	$\overline{\text{Alan}}$ mais pas Alan
Déterministe == Non Déterministe	OUI	NON, mais Décidable ; Ambiguïté Indécidable.		Problème ouvert	OUI	OUI
$W \in L$	O(n)	O(n)	$O(n^3)$	PSPACE	EXPSPACE	Indécidable
$L = \varnothing$	Décidable	PTIME	PTIME	Indécidable	Indécidable	Indécidable
L régulier	O(1)	Décidable	Indécidable	Indécidable	Indécidable	Indécidable
L = A*	Décidable	Décidable	Indécidable	Indécidable	Indécidable	Indécidable
L1 = L2	Décidable	Décidable	Indécidable	Indécidable	Indécidable	Indécidable
$L1 \subset L2$, $L1 \cap L2 = \varnothing$	Décidable	Indécidable	Indécidable	Indécidable	Indécidable	Indécidable
Union, concat, étoile	Fermé	Non clos	Fermé	Fermé	Fermé	Fermé
Intersection	Fermé	Non clos	Non clos	Fermé	Fermé	Fermé
Complémentaire	Fermé	Fermé	Non clos	Fermé	Fermé	Non clos

Alan = « diagonalisation(TM) », $\overline{\text{Alan}}$ = « universal » = Complémentaire(Alan) = Mathison

3.1 - AUTOMATES FINIS : EXEMPLES (1/2)

PROBLEME dans l'entrée, identifier les "mots" , avec leur catégorie lexicale

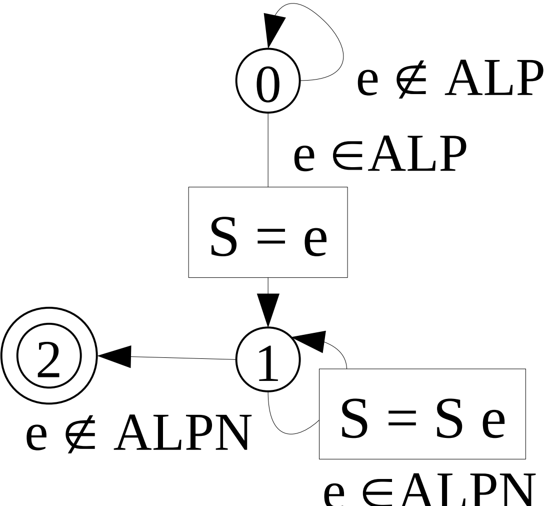
PRINCIPE utiliser des états

répéter {
- *lire le prochain caractère* **e** *en entrée*
- *selon* **e** *et* ***état*** *:*
  - . *changer l'****état***
  - . *actions éventuelles*

} *jusqu'à* <u>*fin de reconnaissance*</u>

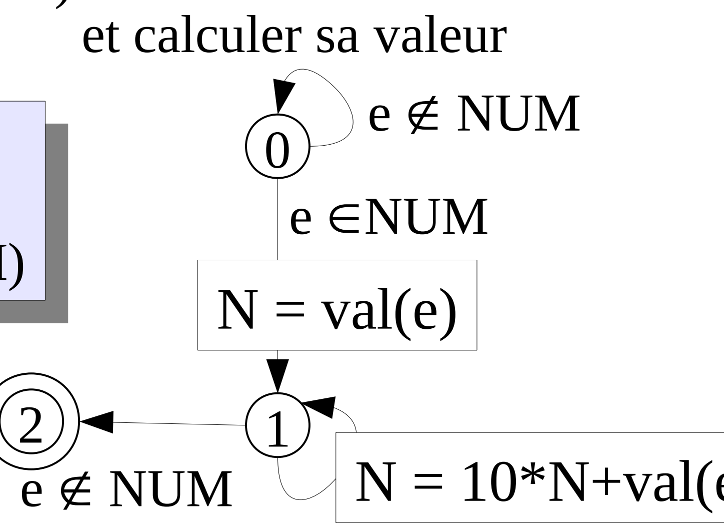
EXEMPLES

a) Reconnaître un symbole de programmation



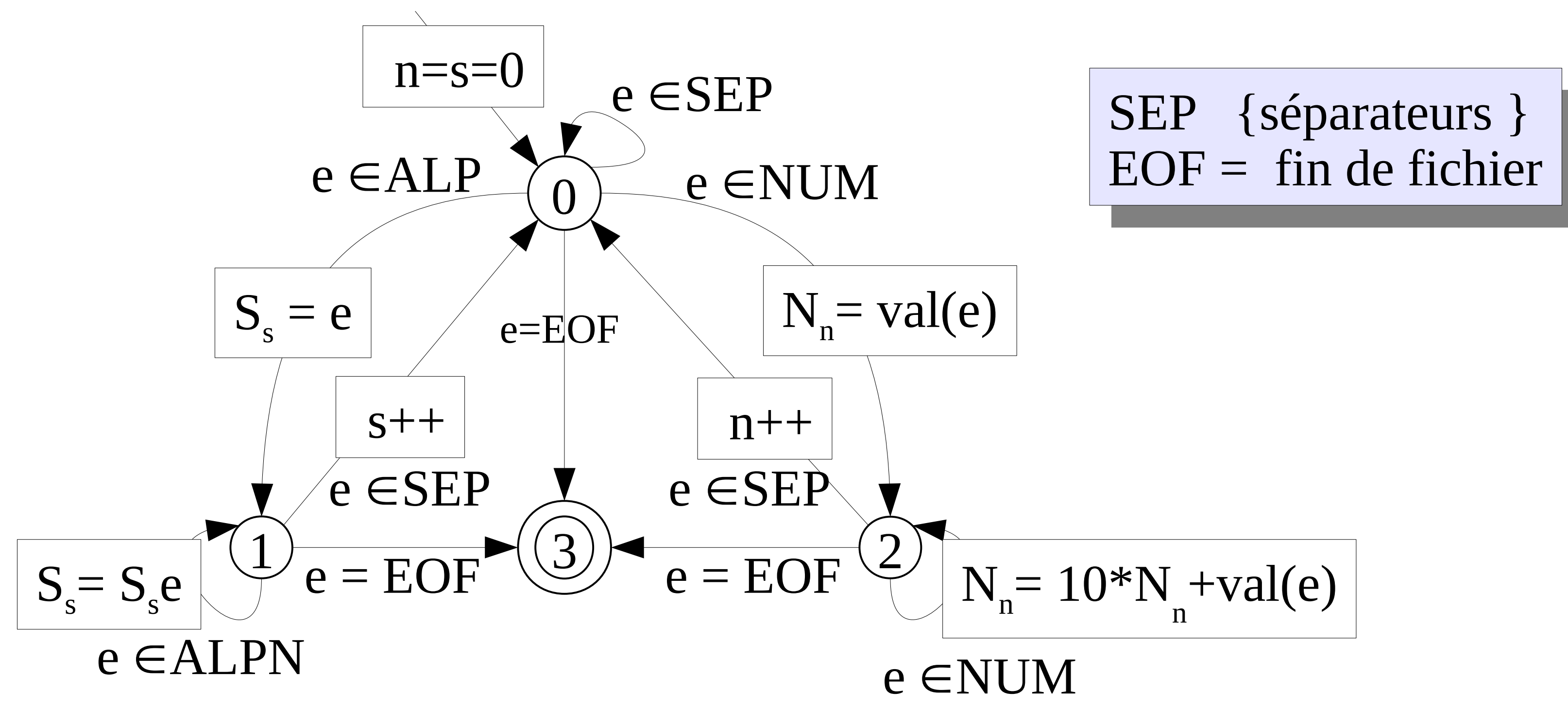
NUM = [0-9]
ALP = [a-zA-Z]
ALPN= (ALP | NUM)

b) Reconnaître un entier et calculer sa valeur

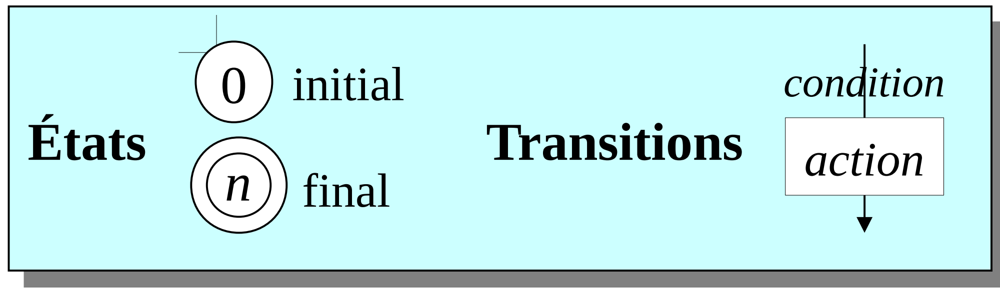


3.1 - AUTOMATES FINIS : EXEMPLES (2/2)

c) Reconnaître tous les symboles de programmation, et tous les entiers



DIAGRAMMES d'AUTOMATES



3.2 - AUTOMATES FINIS : DEFINITION

Automate fini A
$(\Sigma, \delta, Q, q_0, Q^T)$

FSA =Finite State Automaton

- $\Sigma = \{ e_0 , \ldots , e_{m-1} \}$, alphabet des **événements** (caractères lus)
- $Q = \{ q_0 , \ldots , q_{n-1} \}$, **ensemble fini d'états**, dont :
  - q_0 , état **initial** (numéroté 0 ou fléché)
  - $Q^T \subset Q$, états **terminaux** ou "accepteurs"
- δ fonction de **transition** de $Q \times \Sigma$ dans Q,
  - $q = \delta(p , e)$, si transition de p à q sur événement e

SPECIFICATION d'un automate
- par diagrammes d'états
- par tables états/transitions

Table pour l'exemple précèdent :

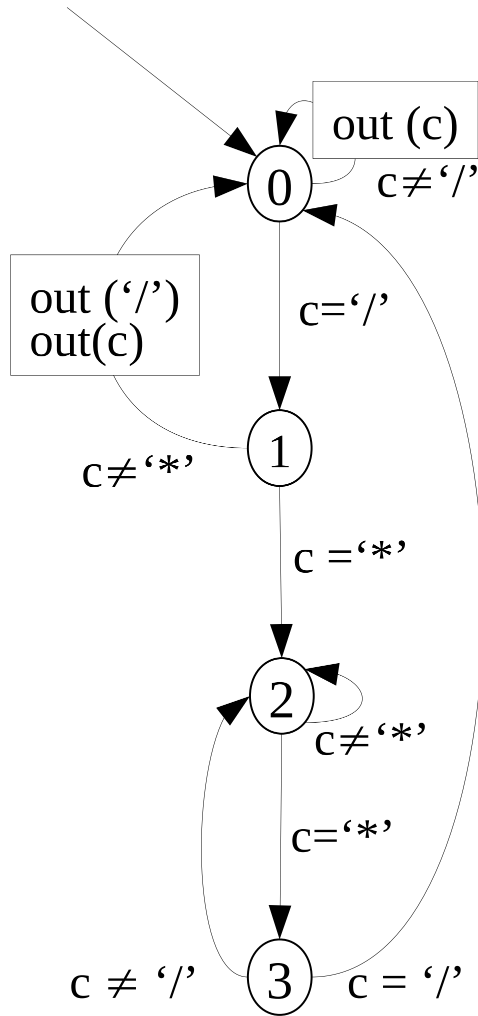
prochain état / action

	$e \in ALP$	$e \in NUM$	$e \in SEP$	$e = EOF$
0	1 / $S_s = S_s$ e	2 / N_n=val(e)	0	3 / Sortie
1	1 / S_s = e	1 / S_s=S_s e	0 / s ++	3 / Sortie
2	0 / erreur	2 / N_n=10 ...	0 / n ++	3 / Sortie

3.3 - Exemple d'application : TRAITEMENT de TEXTE

PRINCIPE

- événement **c** en entrée = caractère lu
- automate
  - → traducteur : génère un texte en sortie
  - → reconnaisseur simple : accepte / rejette



EXEMPLE : Supprimer les commentaires "/* … */" du langage C

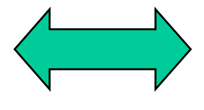
- **Principe d'un automate** (reste à traiter EOF)

- **Reconnaissance par expression régulière**
  . une expression régulière possible : "/*".*"*/"

3.4 – RECONAISSANCE par un AUTOMATE FINI

LANGAGE d'un automate

séquence de symboles $e_1, \dots e_k$
= mot accepté/reconnu par A
=> langage



mène l'automate A de l'état initial q_0
à un état $q_k \in Q^T$ terminal ("accepteur")

EXEMPLE : Reconnaissance de chaînes

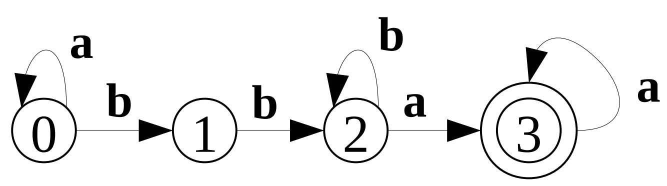
Sur l'alphabet ∑={a, b}, reconnaître une chaîne :

- *contenant une seule chaîne de* **b** *consécutifs*
- *contenant au moins deux* **b**
- *éventuellement précédés d'un ou plusieurs* **a**
- *et obligatoirement suivis d'au moins un* **a**

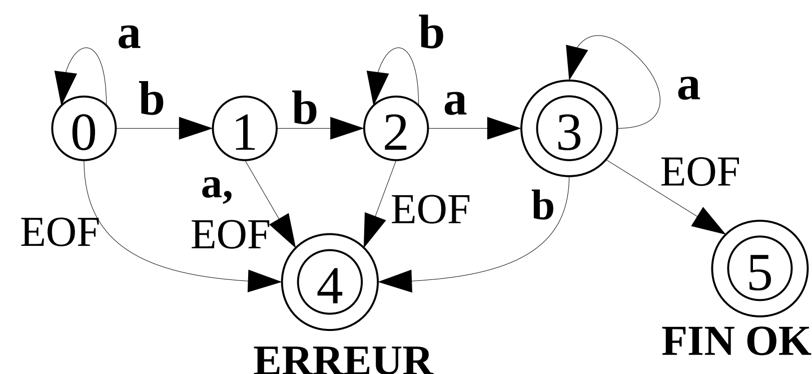
$L=\{a^n b^m a^p, n>=0, m>1, p>0\}$

RegExp = ?

a) sans traitement d'erreur

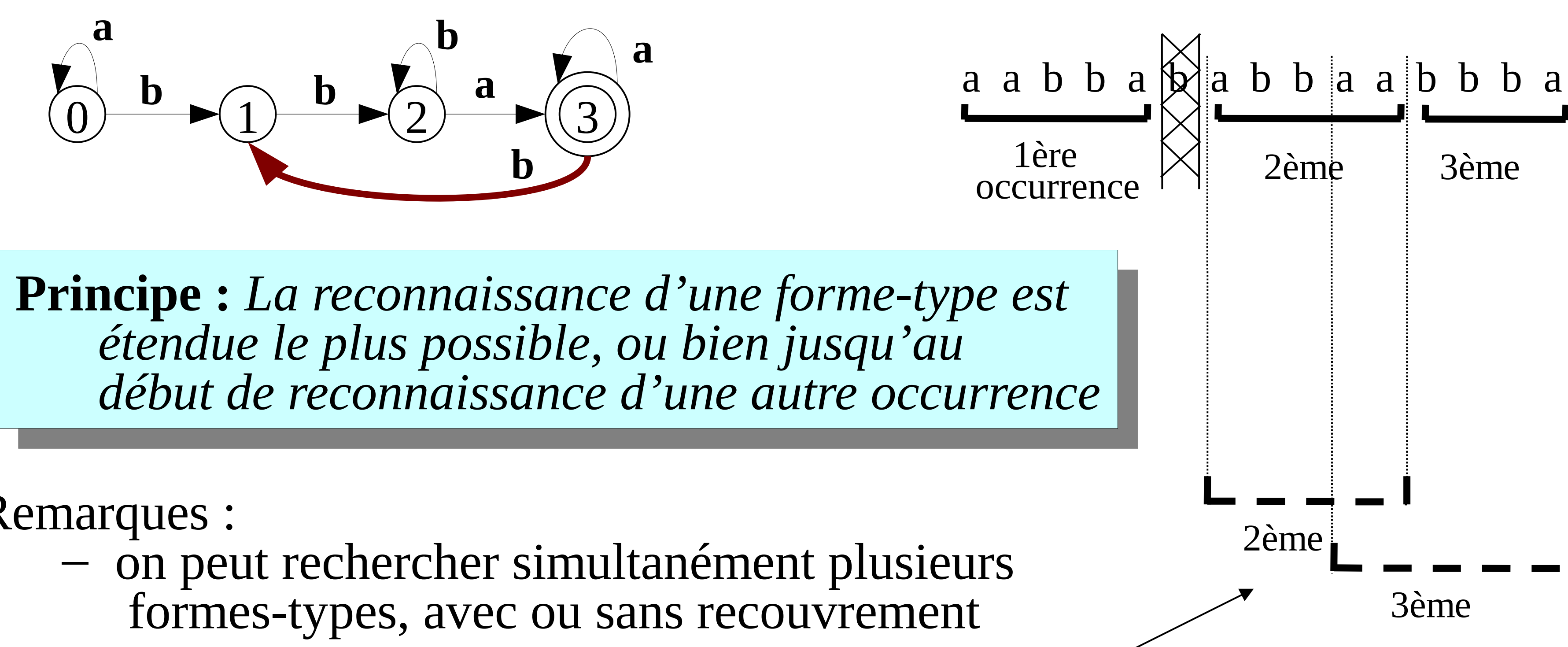


b) avec traitement d'erreur et EOF



3.5 – RECONNAISSANCES MULTIPLES

Reconnaissance de plusieurs occurrences d'une chaîne



Principe : *La reconnaissance d'une forme-type est étendue le plus possible, ou bien jusqu'au début de reconnaissance d'une autre occurrence*

Remarques :

- on peut rechercher simultanément plusieurs formes-types, avec ou sans recouvrement

- il existe des algorithmes reconnaissant toutes les occurrences avec recouvrement

- on peut aussi chercher une forme-type la plus étendue ou la plus réduite : reconnaisseur glouton (*greedy*) ou récalcitrant (*reluctant/ungreedy/lazy*)
  ex: expression "<.*>" sur "<b>bold<\b>" donne <b.....\b> ou <b> ?

3.6 - DETERMINISTE VS NON-DETERMINISTE

DFA ***Deterministic Finite state Automata***	***NFA Nondeterministic Finite state Automata***
- *pour tout couple état-transition* (p,e), *1 seule transition possible* p —e→ q	- ***plusieurs transitions*** *possibles* { p —e→ q_j ; p —e→ q_k ; … } - *ou des* **ε-transitions** *"sans cause"*

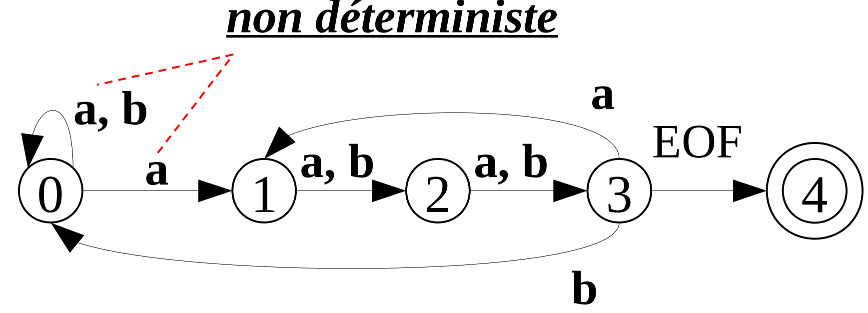
- la reconnaissance n'est pas aléatoire, mais elle doit **explorer toutes les possibilités**

- **Théorème constructif** : **NFA <=> DFA** et il existe des algorithmes de conversion et d'optimisation pour obtenir un DFA équivalent à partir d'un NFA

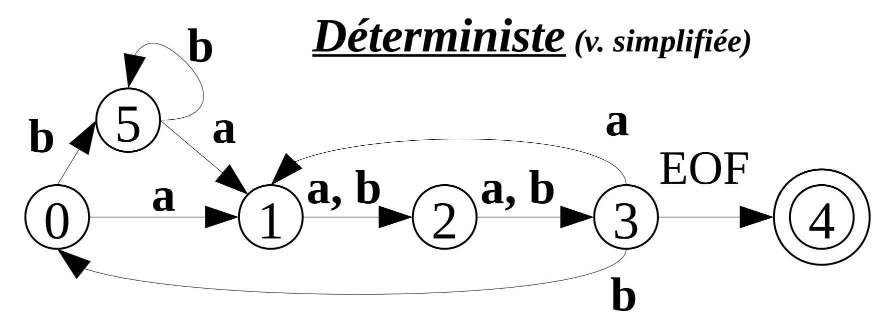
Exemple : Sur l'alphabet ∑={a, b, EOF}, reconnaître une chaîne :

- *contenant au moins un* **a**
- *suivi d'exactement 2 lettres quelconques*
- *le tout avec un éventuel préfixe quelconque*

non déterministe



Déterministe ***(v. simplifiée)***



3.7 - LANGAGE RECONNU PAR UN AUTOMATE

THEOREME (kleene 1956)

Langage Rationnel (RegExp) <=> Langage reconnu par Automate Fini

"Preuve" : cyclique et constructive

(1) RegExp => NFA avec ε-transitions :
constructions union/concaténation/kleene sur automates finis

(2) NFA avec ε-transitions => NFA sans ε-transitions :
algorithmes de fermeture transitive

(3) NFA sans ε-transitions => DFA :
états du DFA = ensemble des parties des états du NFA,
« construction par sous-ensemble »

(4) DFA => Regexp :
récurrence à 3 indices, R(i,j,k) = regexp des mots « passants » de l'état i à l'état j dans l'automate à k états (McNaughton &Yamada)

(5) DFA => DFA minimal :
Différents algorithmes (Nerode, Hopcroft, ...)

Corollaires : - Le complémentaire d'un langage rationnel est rationnel.
- Et donc aussi fermeture par intersection.

Notions sur la TRADUCTION

4) Grammaires

Rappel :

1.3 - Grammaire : définition et utilisation 12
1.4 - Arbre syntaxique 13

4.1 - Exemple de grammaire 31
4.2 - Grammaires ambiguës 33
4.3 - Grammaires régulières 34
4.4 - Grammaires d'opérateurs 36
4.5 - Spécification avec BNF 37

4.1 - EXEMPLE DE GRAMMAIRE (1/2) : spécification

EXEMPLE : ébauche d'un langage informatique

Règles syntaxiques

```
Program        :  Definitions   Functions   ;
Definitions    :  OneTypeVarDefs
               |  Definitions   OneTypeVarDefs
               ;
OneTypeVarDefs :  SIMPLE_TYPE   VarDefList   ';'
               ;
VarDefList     :  VarDef
               |  VarDefList   ','  VarDef
               ;
VarDef         :  ScalarVarDef    |    ArrayVarDef    ;
ScalarVarDef   :  SYMBOL  '='  Number
               |  SYMBOL
               ;
ArrayVarDef    :  SYMBOL   '['   INT   ']'
    … etc …    |  ArrayVarDef   '['   INT   ']
Number         :  INT   |    FLOAT    ;
```

Vocabulaire terminal

```
int   |   char   |   float        return   SIMPLE_TYPE ;
[a-zA-Z] [a-zA-Z0-9]*             return   SYMBOL;
-?[0-9]+                          return   INT;
   … etc …
```

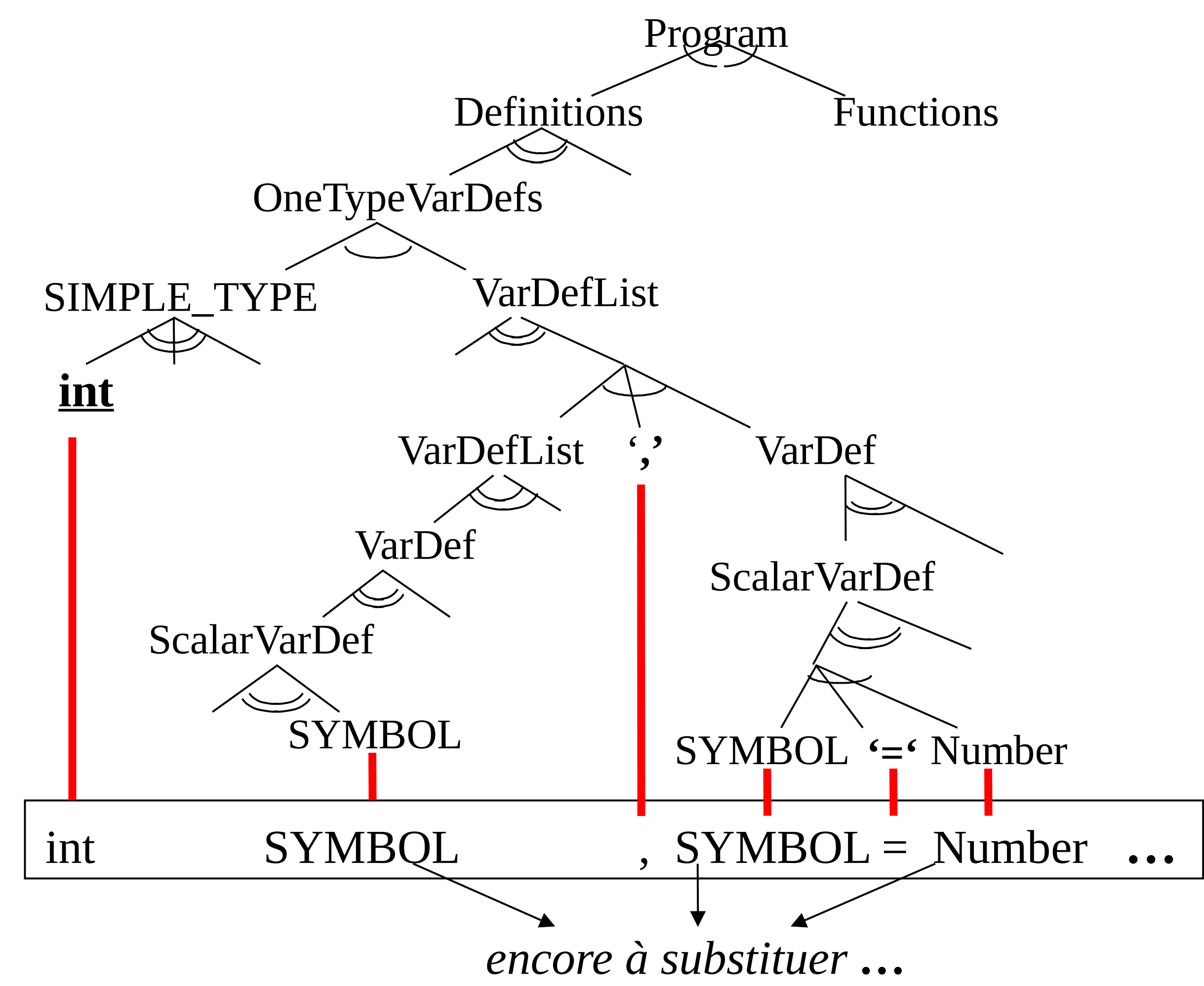
4.1 - EXEMPLE DE GRAMMAIRE (2/2) : utilisation

a) En production

Récursivité des règles

Arbre ET/OU de possibilités

Enumération exhaustive du langage décrit par la grammaire



b) En reconnaissance

- *on cherche à construire **UN** arbre syntaxique pour une phrase donnée*
- *ceci peut nécessiter des retours arrières (alternatives)*

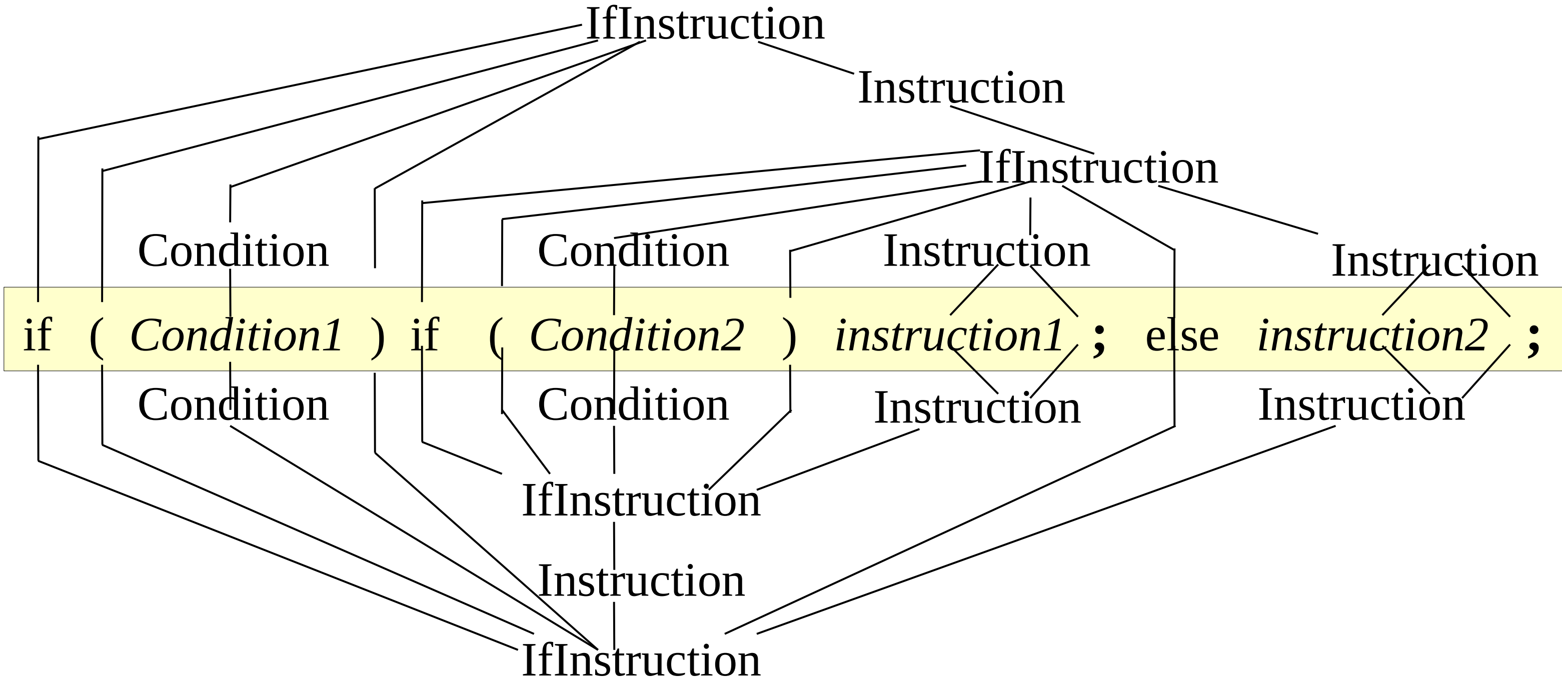
Voir => Analyse syntaxique

4.2 - GRAMMAIRES AMBIGUES

Plusieurs arbres syntaxiques possibles pour au moins **une** chaîne d'entrée

EXEMPLE : instruction if en langage C

```
Instruction    :   IfInstruction |     ….. ‘;’ ;
IfInstruction  :   if  ‘(‘  Condition  ‘)’  Instruction
               |   if  ‘(‘  Condition  ‘)’  Instruction  else  Instruction
               ;
Condition      :   …
```



4.3 - GRAMMAIRES REGULIERES (1/2)

DEFINITION
- toute partie droite de règle contient au plus un symbole non-terminal
- ces symboles figurent en partie droite tous au début, ou tous à la fin

"linéaire à gauche"

$<NT_i>$: $<NT_j>$
| $<NT_k> <T_1> <T_2>$ …

"linéaire à droite"

$<NT_i>$: $<NT_j>$
| $<T_1> <T_2>$ … $<NT_k>$

PROPRIETE *Les langages L(G) reconnus par des grammaires régulières sont les langages rationnels reconnus par des expressions régulières*

EXEMPLE

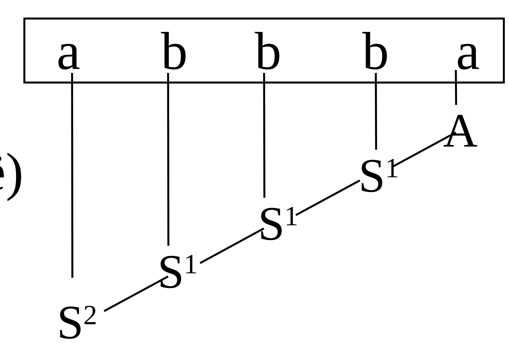
G :
- V_T = {a,b}
- V_N = { S, A}
- Axiome = S

```
S  :  b  A   (S1)
   |  a  S   (S2)
A  :  a
   |  b  A
```

A possibles : a ba bba …

S possibles :
- S1 : ba bba bbba
- S2 : aba abba abbba aaba ...

Exemple d'Arbre Syntaxique (NB : grammaire ambiguë)



L(G) peut être reconnu par l'e.r. : **a * b + a**

4.3 - GRAMMAIRES REGULIERES (2/2)

Les grammaires régulières reconnaissent les langages rationnels

G :

V_N = { Symbole }

V_T = { LETTRE, CHIFFRE }

```
Symbole  :  LETTRE
         |   Symbole  LETTRE
         |   Symbole  CHIFFRE
     ;
```

avec LETTRE = { A,B,…,a,b,… }
CHIFFRE = { 0,1,…, 9 }

L(G) peut être reconnu par l'e.r. : [A-Za-z] [A-Za-z 0-9]+

mais sont une catégorie de grammaires aux possibilités restreintes

G :

V_N = { S }

V_T = { 0, 1 }

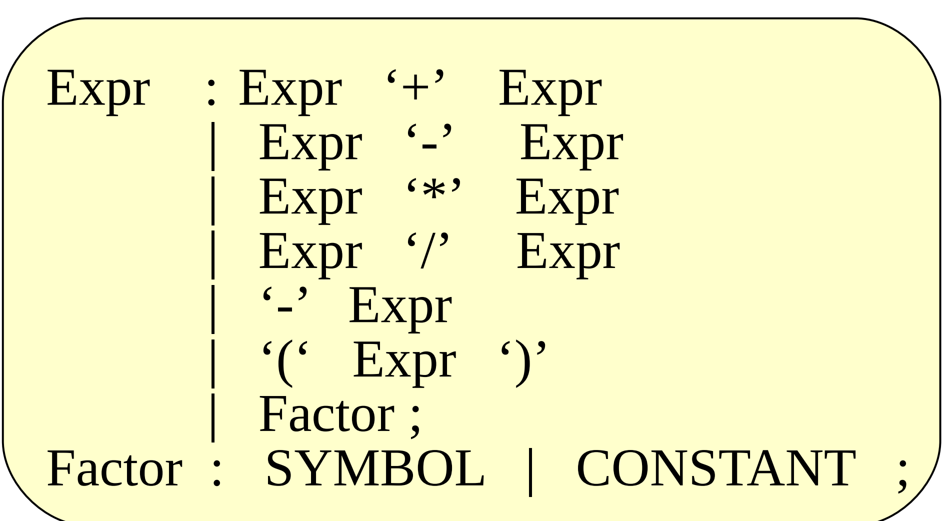
```
S  :  0  S  1
   |   0  1
;
```

- quelques mots de L(G) : 01, 0011, 000111, …
- grammaire non régulière
- pas d'expressions régulières pour L(G)

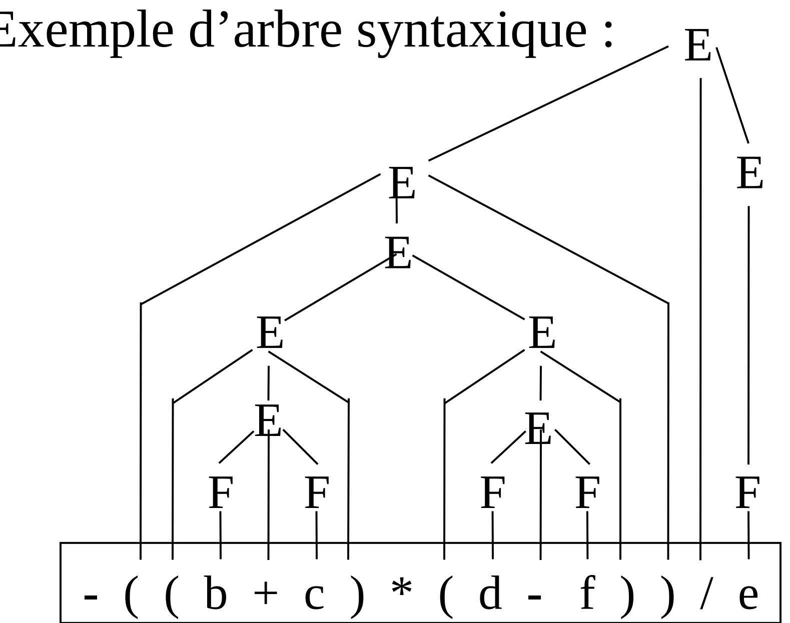
4.4 - GRAMMAIRES D'OPERATEURS

DEFINITION a) aucune partie droite de règle n'est vide
b) ni ne contient 2 symboles non terminaux ***voisins***

EXEMPLE : *Expressions arithmétiques en notation "infixée"*



Exemple d'arbre syntaxique :



Grammaires généralement ambiguës : Ajout de règles de priorités

4.5 – SPECIFICATION AVEC BNF (1/2)

Syntaxes Multiples : BNF, EBNF, ABNF …

Regroupe Règles de syntaxe et Expressions Régulières

- Terminaux = chaînes de caractères constantes
- Répétitions et groupages dans les règles de syntaxe

Éléments de syntaxe

- Définition avec = ou ::= et éventuellement ; en fin
- Non-Terminaux entre <> ou pas
- Terminaux entre " " ou ' ' ou pas
- Groupage entre ( )
- Optionnel (répétition 0 ou 1) entre [ ]
- Répétition Kleene entre { } , ou *terme
- Répétition numérique n*terme, n*mterme, ...
- Commentaires (* … *) ou ; ...
- Alternative | ou /
- Concaténation éventuellement ,

4.5 – SPECIFICATION AVEC BNF (2/2)

EXEMPLES

Expressions Arithmétiques en BNF/EBNF

```
<Expression > ::=  <Terme>  { ( + | - ) <Terme> } ;
<Terme > ::= <Facteur> { ( * | / ) <Facteur> } ;
<Facteur> ::= <Nombre> | <Variable> | "(" <Expression> ")" ;
<Nombre> ::= [-] <Chiffre> {<Chiffre>} ;
<Chiffre> ::= 0 | 1 | ..... | 9 ;
<Variable> ::= <Lettre> { <Lettre> | <Chiffre> | - } ;
<Lettre> ::= a | b | ..... | Z ;
```

ABNF from RFC822

```
date-time = [ day "," ] date time
day     = "Mon"  / "Tue" /  "Wed"  / "Thu" /  "Fri"  / "Sat" /  "Sun"
date    = 1*2DIGIT month 2DIGIT     ; dd mm yy
month = "Jan"  /  "Feb" /  "Mar"  /  "Apr" /  "May"  /  "Jun"
          / "Jul"  /  "Aug" /  "Sep"  /  "Oct" /  "Nov"  /  "Dec"
time    = hour zone               ; hh:mm:ss zzz
hour    = 2DIGIT ":" 2DIGIT [":" 2DIGIT]
zone    = "GMT" / ....... / ( ("+" / "-") 4DIGIT )
DIGIT =  <any ASCII decimal digit>    ; ( decimal 48.- 57.)
```

5) Analyse syntaxique

5.1 - Analyse syntaxique : 3 stratégies 40
5.2 - Analyse descendante 41
5.3 - Analyse ascendante "shift / reduce" .. 44
5.4 - Automate LR(0) 46

5.1 - ANALYSE SYNTAXIQUE : 3 STRATEGIES

Résolution Générale

- Applicable aux langages algébriques non déterministes
- Complexité cubique $O(n^3)$
- Algorithmes (programmation dynamique) : Earley, CYK, Valiant, ...

Analyse descendante ou « LL » ou prédictive

- Applicable à « beaucoup » de langages algébriques déterministes
- Algorithme plus intuitif
- Contraignant sur l'écriture de la grammaire
- Complexité linéaire O(n)
- Outils : « à la main », javacc, ANTLR, ...

Analyse ascendante ou « LR » ou Shift/Reduce

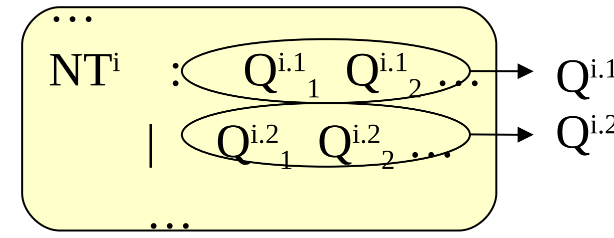
- Applicable à « énormément » de langages algébriques déterministes
- Applicabilité fonction du langage et pas de la grammaire
- Complexité linéaire O(n)
- Algorithmes : Simple LR, LookAhead LR, Canonical LR, …, GLR
- Outils : Bison (LALR ou GLR), GOLD (LALR), ...

NB : GLR = analyse LR généralisée, applicable à **tous** les langages déterministes en O(n),
Et aux langages non déterministes en $O(n^3)$.

5.2 - ANALYSE DESCENDANTE (1/3) : PRINCIPE

1) partir de l'axiome = symbole non terminal NT^1

2) pour reconnaître dans le texte un symbole NT^i,
il y a plusieurs queues de règle possibles $Q^{i.1}$, $Q^{i.2}$, …



- on essaie tour à tour chaque alternative $Q^{i.j}$

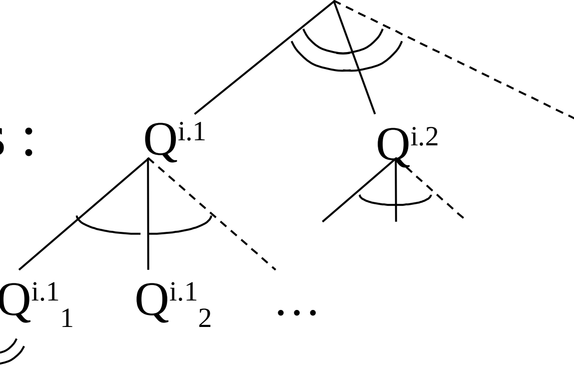
nœud OU

- pour reconnaître $Q^{i.j}$ il faut reconnaître $Q^{i.j}_1$ $Q^{i.j}_2$ …

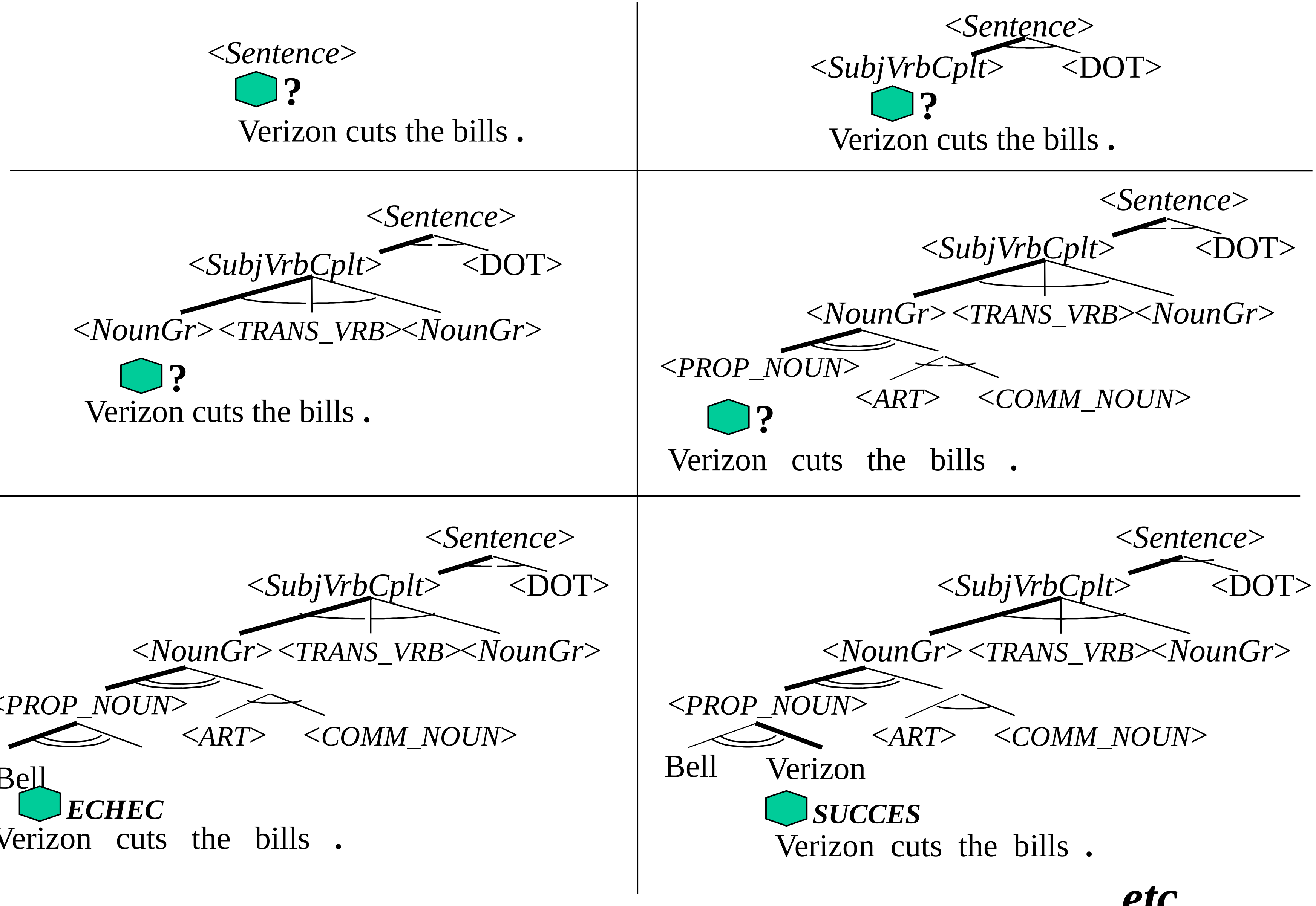
nœud ET

3) exploration de cet arbre ET/OU de possibilités :

- en profondeur d'abord (DFS)
- avec retour arrière si échec



5.2 - ANALYSE DESCENDANTE (2/3) : EXEMPLE



…etc…

5.2 - ANALYSE DESCENDANTE (3/3) : IMPLEMENTATION

- Implémentation possible : par appels récursifs, « descente récursive »

pour reconnaître un symbole non-term. N^i : une procédure Rec_N^i( )
pour chaque forme possible $Q^{i.j}$ de N^i, appeler Rec_$Q^{i.j}{}_1$( ) , Rec_ $Q^{i.j}{}_2$( ) …

- Prédire les choix OU à partir des Tokens suivants (*look-ahead token)*

une règle est applicable si récursivement elle peut générer le prochain Token

- Contrainte : pas de « récursivités gauche » dans la grammaire

directement :

```
Nom :  Nom Lettre
       |  Lettre
```

ou récursivité croisée :

```
A  :  B  ….  ;
B  :  A  …   ;
```

NB : élimination possible par transformation de la grammaire

- D'autres contraintes : factorisation gauche de la grammaire, ...

- Pour approfondir → Algorithmes LL(k)

5.3 - ANALYSE ASCENDANTE "SHIFT/REDUCE"(1/2)

a) on considère une "fenêtre" de mots, étendue progressivement à droite

b) on y essaie des **réductions :** remplacement partie droite -> partie gauche

symbole N_j <= *groupe de symboles* N_{i_k}

REDUCE : *Appliquer une règle dans la fenêtre*

c) sinon on peut lire le token suivant :

SHIFT : *Élargir la fenêtre à la prochaine unité*

d) retour arrière éventuel en cas de blocage

EXEMPLE

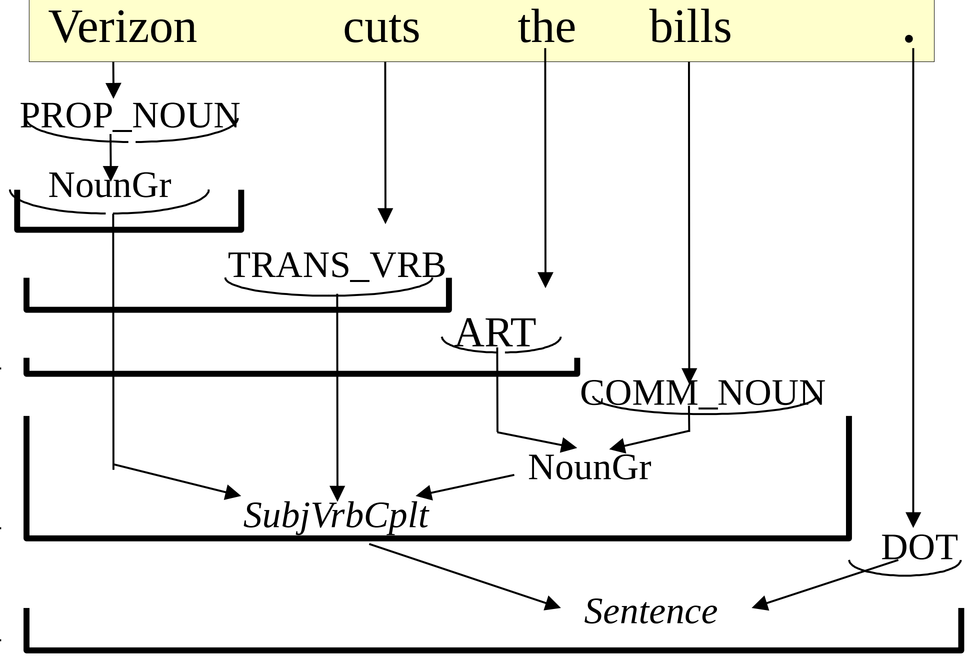
2 REDUCEs, puis SHIFT nécessaire

1 REDUCE, puis SHIFT nécessaire

1 REDUCE, puis SHIFT

3 REDUCEs, puis SHIFT

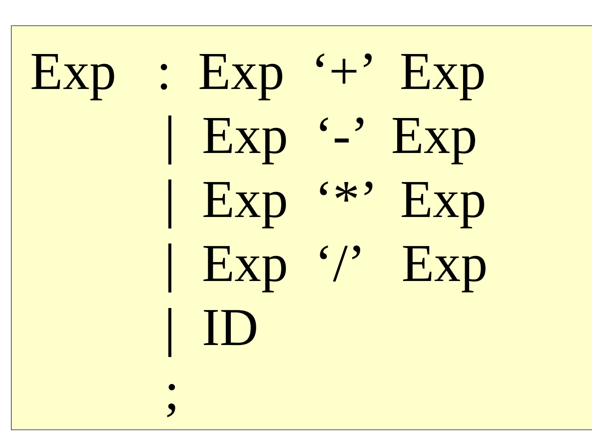
2 REDUCEs, FIN.



5.3 - ANALYSE ASCENDANTE "SHIFT-REDUCE"(2/2)

- on cherche toujours à réduire un **suffixe** de la fenêtre courante
- le contenu de la fenêtre peut être géré comme une **pile**
- méthode adaptée aux grammaires LALR *(Look Ahead, Left-to-right, Right reduce)*

EXEMPLE : grammaire d'opérateurs



Opérations	Pile	Chaîne d'entrée
	\$	\$ ID + ID + ID \$
shift	\$ ID	ID + ID + ID \$
reduce	\$ Exp	+ ID + ID \$
shift	\$ Exp +	+ ID + ID \$
shift	\$ Exp + ID	ID + ID \$
reduce (**)	\$ Exp + Exp	+ ID \$
reduce	\$ Exp	+ ID \$
shift	\$ Exp +	ID \$
shift	\$ Exp + ID	\$
reduce	\$ Exp + Exp	\$
reduce	\$ Exp	\$

(**) Conflit Shift-Reduce, possibilité d'ajouter des règles de priorité pour résoudre l'ambiguïté

5.4 - AUTOMATE LR(0) (1/2)

Décider des opérations (shift, reduce, …) **par un automate finis**

- défini par 2 tables
- l'automate stocke dans une pile :
  - les symboles terminaux intégrés lors d'un shift
  - les symboles non-terminaux remplaçant lors d'un reduce
  - Et pour chaque symbole un numéro d'état associé, repris éventuellement plus tard

Table d'actions A [EtatCourant, SymboleT]

- cellules non vides contiennent des opérations

Sn shift : avancer d'un symbole terminal (token) dans l'entrée

R*k* reduce, en appliquant la k-ième règle $X : A_1 \dots A_{N_k}$

A accept : fin de l'analyse

Sn empiler le token x
associer l'état n à x
se placer dans l'état n

Rk dépiler N_k symboles
e_m état associé au nouveau sommet de pile
prochain état **e'** = G [e_m , X]
empiler X avec état associé associer **e'**
se placer dans l'état **e'**

- cellules vides = cas d'erreur

Table d'états G [EtatRepris, SymboleNT]

- les cellules non vides définissent des changements d'état **g***n* « go to n »

5.4 - AUTOMATE LR(0) (2/2)

Grammaire

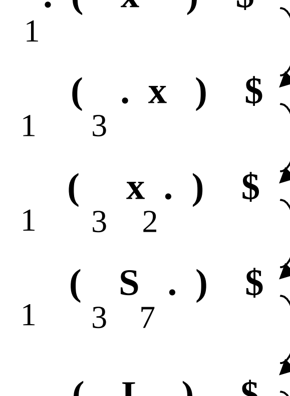
0	S' : S $
1	S : (L)
2	S : x
3	L : S
4	L : L , S

Tables

	A : Actions (	)	x	,	$	G : Chgts d'état S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

Texte (x) $

Analyse



s3 par A [1, ( ] - Empiler (/3 - Nouvel état = 3

s2 par A [3,x] - Empiler x/2 - Nouvel état = 2

reduce par R2, X=S ; Top=3 : par G [3,S], empiler S/7 - Nouvel état = 7

reduce par R3, X=L ; Top=3 : par G [3,L], empiler L/5 - Nouvel état = 5

s6 par A [5,)] - Empiler)/6 - Nouvel état = 6

reduce par R1, X=S ; Top = 1 : par G [1,S], empiler S/4

a par A [4,$]

Notions sur la TRADUCTION

A) Utilisation des outils Flex et Bison

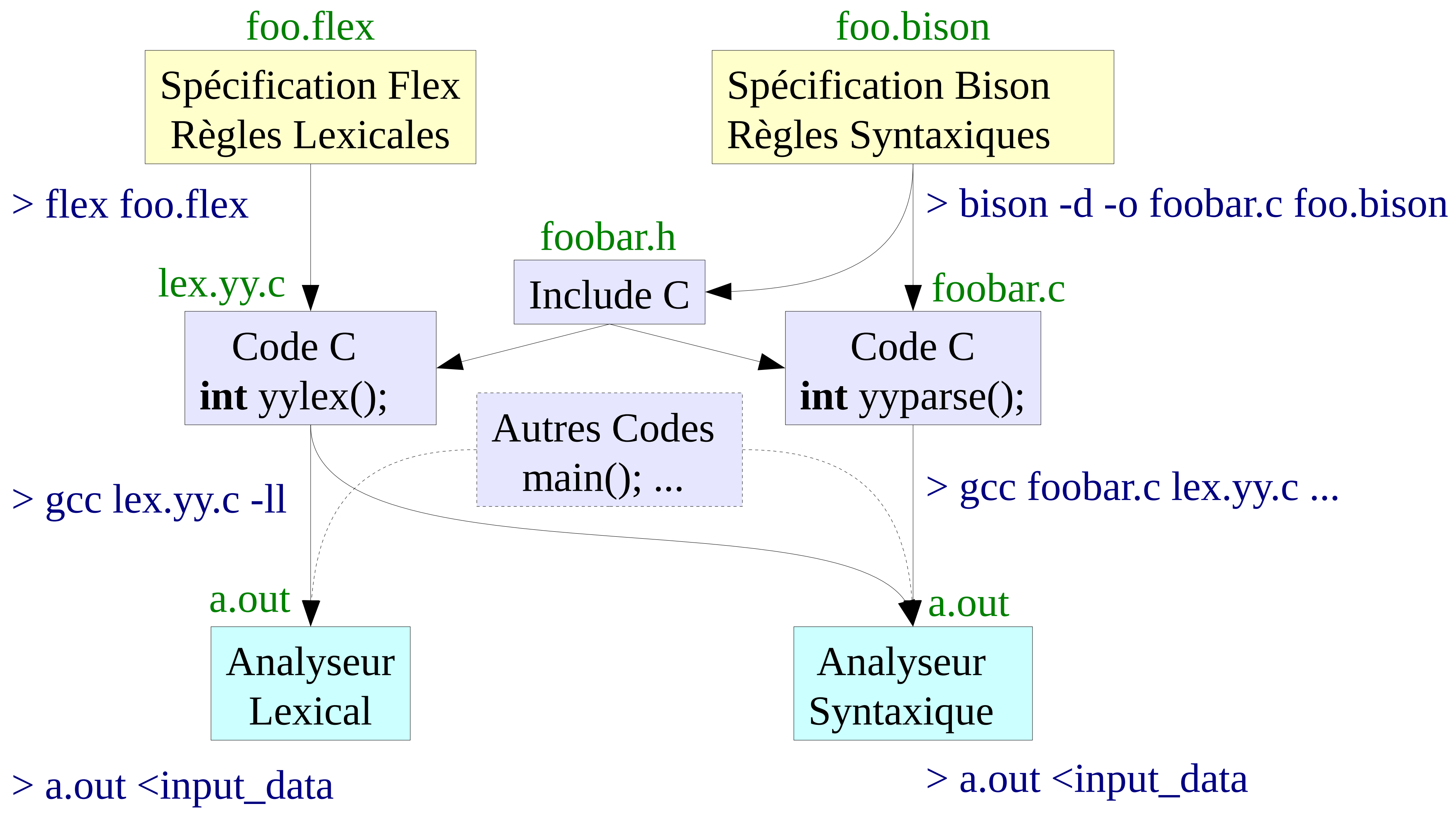
A.1 - Utilisation de Flex et Bison 49
A.2 - Format des spécifications 50
A.3 - Choix du cours et Makefile 51
A.4 - Couplage Flex/Bison : TOKEN 52
A.5 - Couplage Flex/Bison : Valeur de TOKEN 53
A.6 - Valeur de symboles NON-TERMINAUX 54

Cf. mementoFlex.html et mementoBison.html
ou les pages de manuel pour plus de détails

B) Références bibliographiques 55

Cf. version commentée biblio.html

A.1 - UTILISATION DE FLEX ET BISON



A.2 - FORMAT DES SPECIFICATIONS

FLEX		BISON
%{ #include <stdio.h> **void** echo() {printf("hello ");} **int** line=0; %}	Définitions Déclarations C	%{ #include <stdio.h> **extern int** yylex(); **static void** yyerror(); %}
%option noinput 7bit **%s** etat DIGIT [0-9]	Déclarations Flex ou Bison	**%error-verbose** **%token** TOK1 TOK2
%%		%%
Expression { */* bloc Action */* } {DIGIT}* { echo() ; } <etat>[a-z]* ; bleu | vert printf("ciel! ");	Règles et Actions	*symb* : *symb1* {*/*Action1*/*} | *symb* TOK1 {*/*Action2*/*} ; *symb1* : TOK2 {printf("yes! ");} ;
%%		%%
int yywrap (**void**) {return 1;} **int** main(**int** argc, **char** *argv[]) { **while** (yylex()!=0) ; **return** 0 ; }	User Code	**int** yyerror(**char const** *msg) {...} **int** main(**int** argc, **char** *argv[]) {**return** yyparse(); }

A.3 – CHOIX DU COURS ET MAKEFILE

Nommage

foo.flex ---flex--> yylex.c ---gcc--> foo (analyseur lexical)

foo.bison --bison--> yyparse.c yyparse.h ---gcc--> foo (analyseur syntaxique)

Pas de librairies flex/bison (-ll -ly)

Définitions des fonctions : **yywrap()** dans foo.flex,
yyerror() dans foo.bison,
main() dans la spéc adéquat.

Squelette de MAKEFILE

```
foo : EXT_SRC = mes_extras.c -lma_lib     #ajout de sources externes
% :: %.bison %.flex
	flex  -o yylex.c  $*.flex
	bison -d -o yyparse.c $*.bison
	gcc -Wall -I. -o $@  yylex.c yyparse.c $(EXT_SRC)
% :: %.flex
	flex  -o yylex.c  $*.flex
	gcc -Wall -I. -DFLEXALONE -o $@  yylex.c $(EXT_SRC)
```

Cf. fichiers prototypes

A.4 - COUPLAGE FLEX/BISON : TOKEN

OBJECTIF : Partager la fonction yylex() et sa valeur de retour

FLEX

```
%{
#include "yyparse.h"
%}
%%
[a-z][a-z0-9]*
          {return(TOK2);}
[0-9]+  {printf("NB ");
           return(TOK1);}
','       {return(',');}
\n        {return(yytext[0]);}
.         {return(yytext[0]);}
%%
int yywrap (void) {return 1;}
```

yyparse.h ← bison -d

#define TOK1 258
#define TOK2 259

Tokens implicites

0 = EOF
1-255 = char ASCII

BISON

```
%{
extern int yylex();
#include <stdio.h>
static int yyerror(...) {...};
%}
%token TOK1 TOK2
%%
symb : symb1 '\n'
        | symb ',' TOK1
        ;
symb1 : TOK2 {printf("ok");}
%%
int main(void)
     {return yyparse();}
```

A.5 - COUPLAGE FLEX/BISON : VALEUR DE TOKEN

OBJECTIF : Passer une VALEUR sémantique avec un Token

FLEX

```
%{
#include <stdio.h>
#include <string.h>
#include "yyparse.h"
%}
%%
[a-z][a-z0-9]*
    {yylval.valstr=
        strdup(yytext);
     return(TOK2) ; }
[0-9]+
    {sscanf(yytext, "%d",
        &yylval.valint) ;
     return(TOK1) ; }
.   {return(yytext[0]) ; }
%%
int yywrap (void) {...}
```

yyparse.h ← bison -d
typedef union {…}
YYSTYPE;
extern
YYSTYPE yylval;

$i = i^eme symbole term. ou non-term. dans membre droit de la règle

« $i=yylval » réalisé en interne dans yyparse à chaque appel de yylex

BISON

```
%{
extern int yylex();
%}
%union {
      int  valint;
      char *valstr;
}
%token <valint> TOK1
%token <valstr> TOK2
%%
symb : symb1
        | symb TOK1
                    {printf("%d", $2 );}
symb1 : TOK2  {printf("%s", $1 );}
%%
int main(void) {...}
```

A.6 - VALEUR DE SYMBOLES NON-TERMINAUX

OBJECTIF : VALEUR sémantique pour tous les symboles

$i → i[eme] symbole terminal ou non-terminal dans membre droit

$$ → symbole non-terminal du membre gauche

BISON

```
%{
extern int yylex();
%}
%union {
      int  valint;
      char *valstr;
}
%token <valint> TOK1
%token <valstr> TOK2
%type <valint> symb
%%
symb : symb1                {$$ = 0 ; }
        | symb TOK1         {$$ = $1 + $2 ; }
symb1 : TOK2                {printf("%s", $1 ) ; }
%%
int main(void) {...}
```

B - Références bibliographiques (1/2)

Cf. version commentée biblio.html

Mots-clés :

compilation, théorie des langages, expressions régulières, automates, automates finis, grammaires, grammaires algébriques (*context-free*), analyse lexicale et syntaxique, flex et bison (lex et yacc), compilateur de compilateur (*compiler's compiler*), analyse syntaxique ascendante (LR), analyse syntaxique descendante (LL)

1- Bibles

Compilers: Principles, Techniques, and Tools, A. Aho, M. Lam, R. Sethi, J Ulman. 2ieme ed., 2006, Pearson Education, Inc.

Gödel, Escher, Bach: an Eternal Golden Braid, Douglas Hofstadter, 1979, Basic Books

2- Expressions régulières

Mastering Regular Expressions, Jeffrey Friedl, 3ieme ed.. 2006, O'Reilly.

Beginning Regular Expressions, Andrew Watt, 2005, Wrox.

Learning Perl, R.L.Schwarz, T. Christiansen, 1997, O'Reilly.

http://www.regular-expression.info

B - Références bibliographiques (2/2)

3- Bases de théorie des langages

Introduction to Automata Theory, Langages and Computation, John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman. 3ieme ed.2013, Pearson Education.

Introduction to the Theory of Computation, Michael Sipser, 3ieme ed., 2013, Cengage Learning.

Langages Formels, Calculabilité et complexité, Olivier Carton, 1er ed. 2012. Vuibert

Logique et automates P. Bellot, J. Sakharovitch, 1998, Ellipses 1998

4- Compilation : pratique et outils

Algorithmique du texte, M. Crochemore, C. Hancart, T. Lecroq - Vuibert 2001

Compilateurs, D. Grune, H.E. Bal, C.J.H. Jacobs, K.G. Langendoen, Dunod 2002.

Modern Compiler Design in Java, 2nd edition - A.W. Appel - Cambridge, 2002

flex & bison, John Levine, O'Reilly, 2009.

Manuels : Lexical Analysis With Flex , GNU Bison Manual

Lex & Yacc Tutorial, Tom Niemann, ePaperPress.com.

Compiler Construction using Flex and Bison, Anthony A. Aaby, Open Publication License. 2004