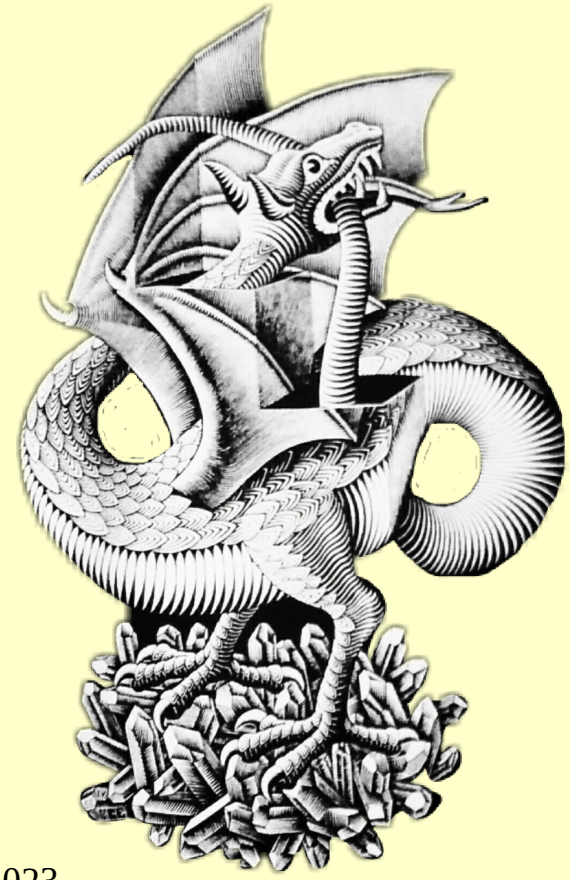
COMPILATION
De l’algorithme à la porte logique



Télécom SudParis

CSC4251-4252

Pascal Hennequin

Sommaire (1/2)

Sommaire (2/2)

Compilation

Chapitre 0

0) Prolégomènes

Compilation

De l’algorithme à la porte logique

Entrée : un algorithme implanté sous forme d’un programme dans un langage de programmation «évolué »

COMPILATEUR

Sortie : un programme équivalent exécutable par un processeur

Exemple : C avec gcc

Interprétation

Autre possibilité

Entrée : un algorithme implanté sous forme d'un programme dans un langage de programmation «évolué »

INTERPRÉTEUR

Sortie : exécution directe

Exemple : Bash, Python

Méli-mélo

On peut mélanger

Entrée : un algorithme implanté sous forme d' un programme dans un langage de programmation «évolué »

COMPILATEUR

Forme intermédiaire : un programme équivalent dans un autre langage «moins évolué »

INTERPRÉTEUR

Sortie : exécution de la forme intermédiaire par un programme interpréteur ou machine virtuelle

Exemple : Java

Traduction

De façon plus large

Entrée : un fichier définissant des objets informatiques (exécution, données, …) en utilisant un format ou syntaxe ou langage

TRADUCTEUR

Sortie : un fichier équivalent (ou pas) utilisant un autre langage

Exemples : Traitement de texte, Conversion d'images, Impression de documents, …, …, PrettyPrint, ...

Phases de Compilation (1/4)

I) Analyse Lexicale

Reconnaître les mots du langage en entrée

πR2=alpha+1

πR2	=	alpha	+	1
IDENT	AFF	IDENT	OPBIN	ENTIER

II) Analyse Syntaxique

Reconnaître les phrases en entrée : construire et valider la structure grammaticale

Grammaire :
R1 : ENTIER → exp
R2 : IDENT → exp
R3 : IDENT AFF exp → phrase
R4 : exp OPBIN exp → exp

R3 (R4 (R2 , R1))
=> phrase correcte

Phases de Compilation (2/4)

III) Analyse Sémantique

Établir la signification du fichier en entrée

Construire les éléments de contexte :

- liaison entre les définitions et les utilisations des variables ou fonctions
- vérification des contraintes de typage, gestion du transtypage
- analyse des chemins d'exécutions, i.e code mort, *return* absent
- ...

πR2=alpha+1

int alpha

double alpha

String alpha

?

Addition entière	Résultat entier ou transtypage avant affectation
Addition flottante	Transtypage (double) 1 , ...
Concaténation chaines	Transtypage 1.toString() , Méthode String.append(), ...

Phases de Compilation (3/4)

IV) Code Intermédiaire

Traduction de l'entrée dans un langage intermédiaire qui se veut :

- Indépendant du langage en entrée et plus simple
- Indépendant du langage en sortie ou du traitement à suivre

Interface entre **partie avant** et **partie arrière** du compilateur

```
πR2=alpha+1
```

```
int πR2;
void F(int alpha) { πR2=alpha+1; }
```

```
Code à 3 adresses :
t2:=alpha PLUS 1
πR2:=t2
```

```
bytecode Java :
1    iload_1 [arg0]
2    iconst_1
3    iadd
4    putfield Test2.πR2 : int [2]
```

Phases de Compilation (3 bonus/4)

πR2=alpha+1

```
String πR2;
void F(String alpha) { πR2=alpha+1; }
```

```
bytecode Java (cas String)
1   new java.lang.StringBuilder [2]
4   dup
5   invokespecial java.lang.StringBuilder() [3]
8   aload_1 [arg0]
 9  invokevirtual java.lang.StringBuilder.append(java.lang.String) : java.lang.StringBuilder [4]
12 iconst_1
13 invokevirtual java.lang.StringBuilder.append(int) : java.lang.StringBuilder [5]
16 invokevirtual java.lang.StringBuilder.toString() : java.lang.String [6]
19 putfield Test2.πR2 : java.lang.String [7]
```

Phases de Compilation (4/4)

V) Génération de code

Optimisation du code intermédiaire

Génération du code cible

Optimisation du code cible

Production de l'exécutable et édition de liens

```
MIPS :
move $t2, $a1
li $v1, 1
add $t2, $t2, $v1
sw $t2, 0($a0)
```

```
ELF (gcc) :
4004f7: mov    %rsp,%rbp
4004fa: mov    -0x4(%rbp),%eax
4004fd: add    $0x1,%eax
400500: mov    %eax,0x200b2a(%rip)
```

V bis) Interprétation

Exécution directe du code intermédiaire

Objectif du Cours (1/2)

Écrire un compilateur pour un sous-ensemble du langage Java vers le langage assembleur MIPS

- Comprendre par la pratique les enjeux et les techniques des différentes phases de la compilation

Trois objectifs sous-jacents

- Analyse lexicale et syntaxique : maîtriser les concepts théoriques et leurs mises en œuvre dans les outils « *compiler's compiler* »
- Architecture des processeurs : instructions, registres, structures algorithmiques, appel de fonction, gestion mémoire ...
- Algorithmique et programmation :
  - Dans l'écriture du compilateur (algorithmique d'arbre,...)
  - Dans la compréhension de la chaîne de compilation et de la façon dont les paradigmes usuels de programmation sont implantés pour l'exécution

Objectif du Cours (2/2)

Ne fait pas partie du cours

- Analyse sémantique dynamique et analyse de flux
- Aspects théoriques de l’analyse sémantique
- Optimisation du code généré et transformation d’arbre
- Optimisation du code du compilateur
- Compilation séparée et édition de lien

Bibliographie (1/2)

Mots clés

Compilation, théorie des langages, expressions régulières, automates finis, grammaires algébriques (*context-free*), compilateur de compilateur (*compiler's compiler*), analyse syntaxique ascendante (LR)

Bibles

- **Gödel, Escher, Bach: an Eternal Golden Braid,** aka "GEB", Douglas Hofstadter, 1979, Basic Books
- **Compilers: Principles, Techniques, and Tools**, aka «dragon book», A. Aho, M. Lam, R. Sethi, J Ullman. 2nd ed., 2006, Pearson Education, Inc.
- **Modern Compiler Design in Java**, 2nd edition - A.W. Appel - Cambridge, 2002
- **Compilateurs**, D. Grune, H.E. Bal, C.J.H. Jacobs, K.G. Langendoen, Dunod 2002.

Bibliographie (2/2)

Références

- **Introduction to Automata Theory, Langages and Computation**, John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman. 3ieme ed.2013, Pearson Education.
- **Langages Formels, Calculabilité et Complexité**, Olivier Carton, 1er ed. 2012. Vuibert
- http://www.regular-expression.info
- **Learning Perl**, R.L.Schwarz, T. Christiansen, 1997, O'Reilly.
- **Mastering Regular Expressions**, Jeffrey Friedl, 3ieme ed.. 2006, O'Reilly.

Chapitre 1

1) Analyse Lexicale

De la théorie (1/2)

Théorie des Langages, Théorie de la Calculabilité

Que peut on calculer automatiquement ?

Que peut on calculer efficacement ?

Systèmes et grammaires formels

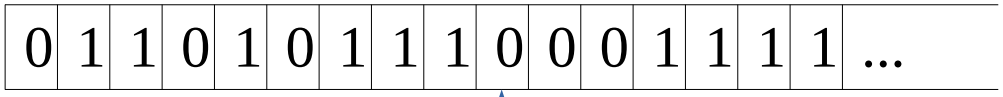
Hiérarchie de CHOMSKY-SCHÜTZENBERGER

Type	Langage / Grammaire	Machine
0	Récursivement énumérable	Machine de Turing
1	Contextuel (*context-sensitive*)	Automate à ressources bornées
2	Algébrique (*context-free*)	Automate à pile
3	Rationnel (*regular*)	Automate fini

De la théorie (2/2)

Machine de Turing et Calculabilité

BANDE : infinie



TÊTE : Lecture / Écriture 0 ou 1
Déplacement Droite ou Gauche

CONTRÔLE :

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

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
- Théorème de Rice : toute propriété non triviale est indécidable

Et en Pratique

Langage de type 3 = Expressions Régulières

- Reconnues par automates à états finis
- ∃ outils pour reconnaître les expressions régulières en temps O(n)
  - Générateurs d'analyseur lexical : lex, flex, JFlex …
  - Aussi : commande grep, langage Perl, éditeurs de texte, …, ...

Langage de type 2 = Grammaires Algébriques

- Reconnues par automates à Pile
- Couvre l'ensemble des langages informatiques
- ∃ outils pour faire l'analyse syntaxique en temps O(n)
  - yacc (*Yet Another Compiler's Compiler*), Bison, CUP, ...
  - ANTLR, JavaCC, ...

Lexicale versus Syntaxique (1/2)

Séparation usuelle dans les langages naturels

- Les mots et leur catégorie (nom, verbe, adjectif,..) définis par un dictionnaire
- Les phrases définies par des règles de construction (grammaire)

Séparation théorique dans la hiérarchie de Chomsky

- Expressions Régulières versus Grammaire algébrique

Vocabulaire = Unité lexicale = Lexème

- Les mots autorisés dans le langage

Catégorie Lexicale = Symbole terminal = *Token*

- Ensemble de mots ayant le même comportement dans la grammaire

Grammaire

- Ensemble de règles définissant les séquences de *Token* valides

Lexicale versus Syntaxique (2/2)

Exemple : langage naturel élémentaire

- Lexique

```
POINT     =  .
ARTICLE   =  un | le
VERBE     =  mange |  tue
NOM_C     =  lapin | chasseur | gnou
NOM_P     =  Bart | Jeannot | Obi-Wan
```

- Syntaxe

```
<Phrase>          := <SujetVerbeCompl>  POINT  ;
<SujetVerbeCompl> := <GroupeNom>  VERBE  <GroupeNom> ;
<GoupeNom>        :=  NOM_P  |  ARTICLE  NOM_C ;
```

- Des phrases valides :
  - Obi-Wan mange un gnou.
  - le lapin tue un chasseur.

Analyse lexicale (1/3)

Reconnaître

- Les mots du langage, ou unités lexicales ou lexème
- Leur catégorie lexicale ou token

Qui est lexème ?

- Une chaîne de caractères dont la signification n’est pas construite à partir de ces composants.
- Exemples : Mots clés (for while if) , Identificateur (ma_variable forif), Opérateurs (+ = ++ +=), Ponctuation (; , .), Commentaire (/* coucou */)
- Contre-exemple : une constante entière est en général un lexème même si conceptuellement la numération est un processus syntaxique

Analyse lexicale (2/3)

Qui est catégorie lexicale ?

- Ensemble de lexèmes non différentiés dans les règles de grammaire
- Dépend de l'écriture de la grammaire
  - OpBin={+ , - , * , / } ou OpPlus, OpMoins, OpMult, OpDiv ?

Token

- Entier identifiant la catégorie lexicale reconnue (type énuméré)
- De façon optionnelle :
  - Valeur du lexème dans la catégorie pour les traitements après l'analyse syntaxique
  - Position du lexème dans le fichier source
  - ...

Analyse lexicale (3/3)

Analyseur Lexical (*lexer, tokenizer*)

- **Entrée** : Texte ASCII, ou Unicode, ou éventuellement binaire

- **Sortie** : une séquence de symboles terminaux (t*okens*) qui servira d'entrée à un analyseur syntaxique.

Exemple

- Entrée : Alpha += 32 * bêta + 2 /* comment*/

- Sortie : 1 3 2 4 1 4 2

Catégories={ Ident., Littéral, Affect., OpBin}

- Ignore : blancs, commentaires, ...

Spécification du Vocabulaire

Par énumération

- Vocabulaire fini
- Structure de dictionnaire

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

Par règle de production récursive

- Vocabulaire infini
- Reconnaissance complexe
- Ex : BNF

```
<Entier> :=  <Chiffre>
           |  <Entier > <Chiffre> ;
<Nom>    := <Lettre>
           || <Nom > <Lettre> ;
<Chiffre> := '0'  |  '1' |  … | '9' ;
<Lettre>   := 'a' |  'b' |  …   ;
```

Par expression régulière

- Vocabulaire infini
- Formalisme simple
- Reconnaissance automatisable et efficace

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

Expression Régulière (1/3)

Définition

Constantes	**ε** Caractère	Mot vide Singleton
3 Opérateurs	$r_1 \mid r_2$ $r_1\ r_2$ r *	Alternative ou Union Concaténation (implicite) Fermeture de Kleene : répété N ≥ 0 fois
Parenthèses	**( )**	Gestion des priorités

Exemple

a* a (b | c)* a* sur ∑ = { a, b, c}

*Mots commençant par au moins un **a**, suivi de **b** ou **c** en nombre quelconque, et éventuellement terminé par des **a***

a, aa, ab, ac, aaa, aab, aac, aba, abb, abc,, abba, ...

Expression Régulière (2/3)

Classes de caractères : Reconnaît 1 symbole parmi un ensemble

.	Un caractère quelconque sauf fin de ligne
[xyzT**]**	Un caractère de l'énumération == x|y|z|T
[… A-F … **]**	Idem avec intervalle
[^…]	Un caractère hors de l'énumération

Répétitions

r **?**	*r* répétée 0 ou 1 fois == *r* | ε
r **+**	*r* répétée n>0 fois == *r r**
r **{k}**	*r* répétée **k** fois
r **{k,l}**	*r* répétée entre **k** et **l** fois

Délimiteurs de contexte

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

Expression Régulière (3/3)

Exemples

[aeiouy]	Une voyelle
[A-Za-z]+	Un mot alphabétique non vide
^[\t]+$	Une ligne blanche
//.*	Un commentaire C++
\"[^"]*\"	Une constante chaîne
[0-9]+ (\.[0-9]*)? ([eE][+-]?[0-9]+)?	Une constante flottante
0 | [1-9] ([0-9_]* [0-9])?	Une constante entière décimale Java

Chapitre 2

2) Automate Fini

Abrégé de la théorie des langages (1/2)

Définitions

Alphabet $\sum$: ensemble fini de lettres

Mot w sur $\sum$: suite finie de lettres de $\sum$

$\sum^*$: ensemble de tous les mots = monoïde libre

Langage L : ensemble de mots inclus dans $\sum^*$

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

Opérations

Alternative (union) , Concaténation (produit),

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

Aussi Intersection, Complémentaire, Quotient, …

Questions

$w \in L$? , $L = \varnothing$?, L fini ?, $L_1 = L_2$?, ...

décidable, calculable, facilement calculable ?

Abrégé de la théorie des langages (2/2)

Langage	Rationnel	Algébrique déterministe	Algébrique	Contextuel	Récursif	Récursivement énumérable
Machine	Automate fini	Automate à pile		Automate borné	Turing qui s'arrête	Machine de Turing
Spécification	Expression régulière	Grammaire *context-free*		Grammaire *context-sensitive*	Grammaire générale ou *phrase-structure*	
Exemples	a^p et p paire	a^nb^n $a^nb^nc^pd^p$ Dyck ([]())	Palindrome $a^nb^nc^p \cup a^pb^nc^n$ a^p et p carré	Papou $a^nb^nc^n$, $a^nb^pc^nd^p$ a^p et p premier	diag(CSL) Ackermann	$\overline{\text{diag(TM)}}$ pas diag(TM)
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 rationnel	OUI	Décidable	Indécidable	Indécidable	Indécidable	Indécidable
$L = \varnothing$	Décidable	PTIME	PTIME	Indécidable	Indécidable	Indécidable
$L = \sum^*$	Décidable	Décidable	Indécidable	Indécidable	Indécidable	Indécidable
$L_1 = L_2$, $L_1 \subset L_2$	Décidable	Décidable	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

Automate fini (1/3)

Principe

```
int état =0 ;
tant que pas_fini {
        lire un caractère c
        selon état et c
                changer état
}
```

Automate Fini (*Finite State Automaton*)

$A = (\sum, Q, q_0, Q^T, \delta)$ avec

$\sum = \{c_0 , \ldots , c_{m-1}\}$, alphabet d'entrée (ou **événements**)

$Q = \{q_0 , \ldots , q_{n-1}\}$, ensemble fini d'**états**

q_0 **état initial**

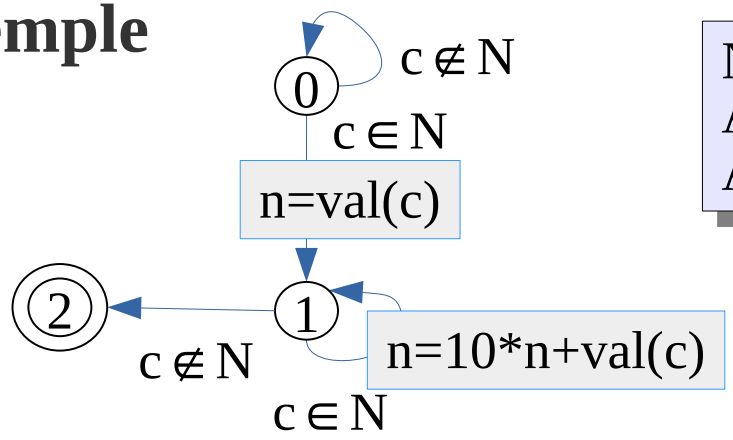
$Q^T \subset Q$, ensemble des états **terminaux** (ou **accepteurs**)

$\delta : Q \times \sum \rightarrow Q$, **fonction de transition**

$q = \delta(p , c)$: transition de p à q sur lecture de c

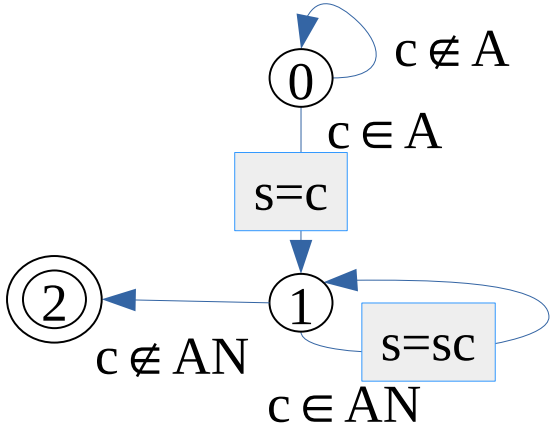
Automate fini (2/3)

Exemple

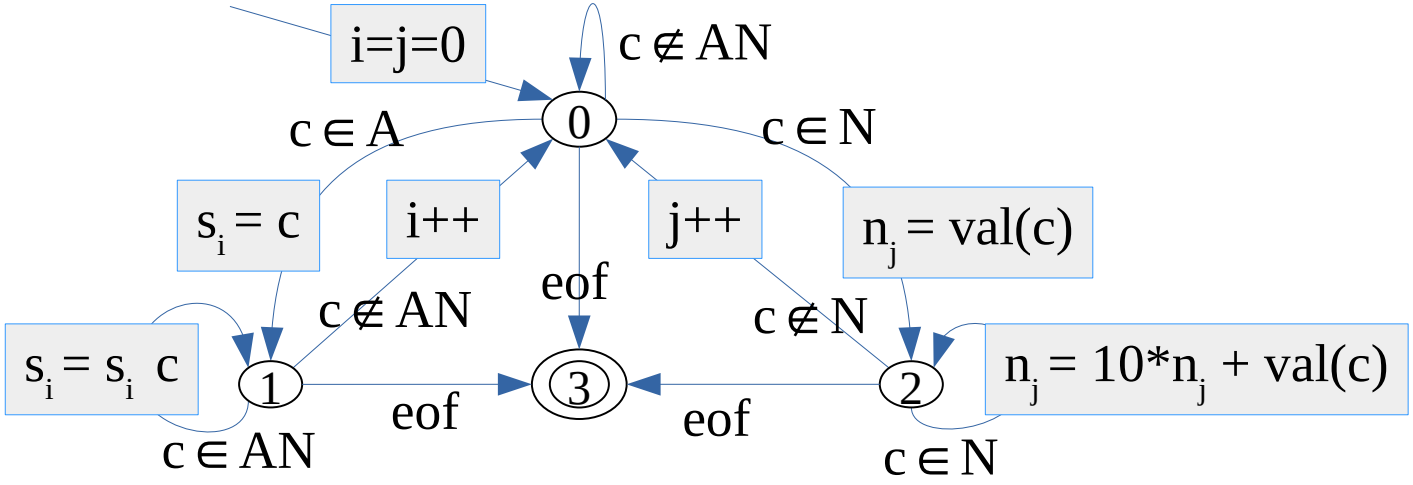


N = [0-9]
A = [a-zA-Z]
AN= A ∪ N

a) reconnaître un entier



b) reconnaître un identificateur

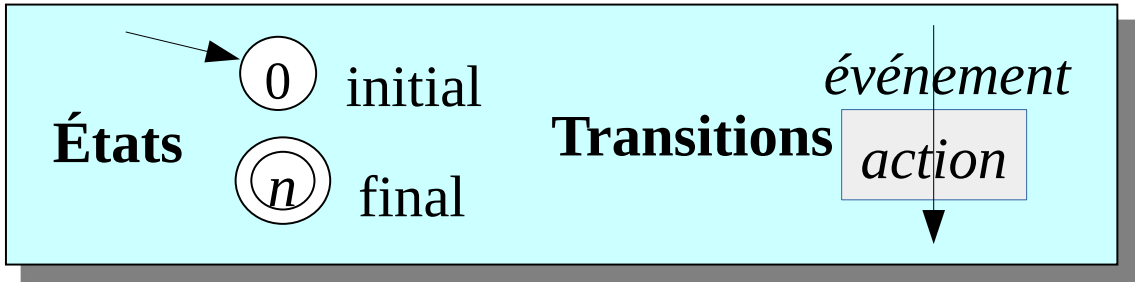


c) Reconnaître tous les identificateurs et tous les entiers

Automate fini (3/3)

Représentations d'un automate

- Diagramme d'état



- Table états/transitions

car. lu / état	c ∈ A	c ∈ N	c ∉ AN	c=EOF
0	1	2	0	3
1	1	2	0	3
2	0	2	0	3

Reconnaissance par un automate

Langage reconnu par un automate

Les mots dont la séquence de caractères mène l'automate de l'état initial q_0 à un état terminal $q \in Q^T$

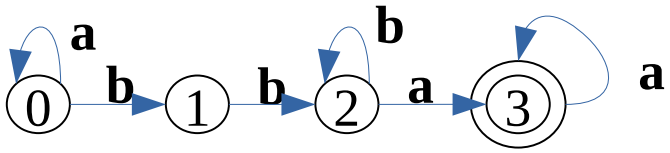
Exemple

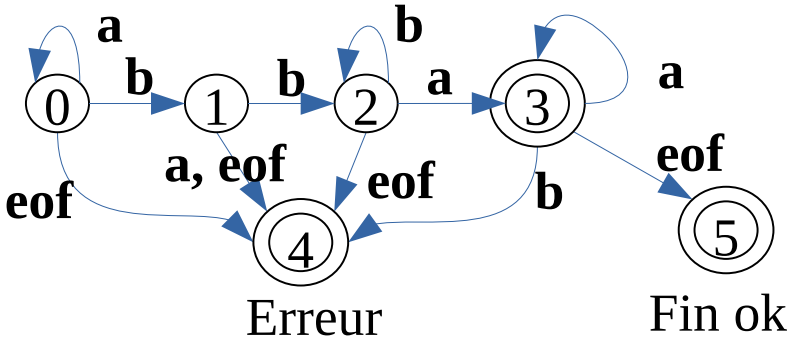
Sur l'alphabet $\sum = \{\mathbf{a}, \mathbf{b}\}$, reconnaître une chaîne

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

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

Expression Régulière ?





Reconnaissance par des outils

Reconnaissance multiple de chaînes valides dans un texte

- Glouton versus Récalcitrant
  - retourner la forme la plus étendue (glouton, *greedy*) ou la plus courte (récalcitrant, *reluctant*, *lazy*)
  - [0-9]+ sur « 1234 » ?
  - < .* > sur « bold » ?
- Avec ou sans recouvrement (aussi : avec ou sans trou)
  - a*bb+a+ sur « abbaabba » ?
    - « abb<u>aa</u> » et « <u>aa</u>bba » ou sans recouvrement : « abbaa » et « bba »
- Plusieurs automates simultanément
  - [0-9]+ et [a-zA-z]+ en même temps

Il existe des algorithmes pour outiller l'automate de base afin d'avoir les différents comportements

Déterministe versus Non Déterministe

Machine non déterministe

Déterministe	Non déterministe
∀ couple (p,c), ∃! transition p -c-> q	- Plusieurs transitions possibles p -c-> q1, p -c-> q2 - ou des ε-transitions (sans caractère lu) p --> q
Exécution linéaire	Exécution par exploration de toutes les possibilités. Un mot est reconnu si il existe un chemin vers un état terminal.

Théorème

Pour tout automate fini non déterministe (NFA), il existe un automate fini déterministe (DFA) reconnaissant le même langage

Langage Rationnel et Automate Fini (1/2)

Théorème de Kleene (1956)

Expressions Régulières <==> Automates Finis

Preuve : cyclique et constructive

1) Regexp → NFA avec ε-transition

   Traduction concaténation/union/étoile sur les automates

2) NFA avec ε-transition → NFA sans ε-transition

   Algorithme de fermeture transitive

3) NFA sans ε-transition → DFA

   « construction par sous-ensemble »

   états du DFA = ensemble des parties des états du NFA.

4)DFA → Regexp

   Récurrence à 3 indices (McNaughtom & Yamada)

Langage Rationnel et Automate Fini (2/2)

Théorème de minimisation

Il existe des algorithmes (Nérode, Hopcroft, …) pour construire des automates déterministes de taille minimale

=> 3bis) DFA → DFA minimal

Corolaire pratique

Outils de reconnaissance par expressions régulières = 1 + 2 + 3 + 3bis

Théorèmes faciles

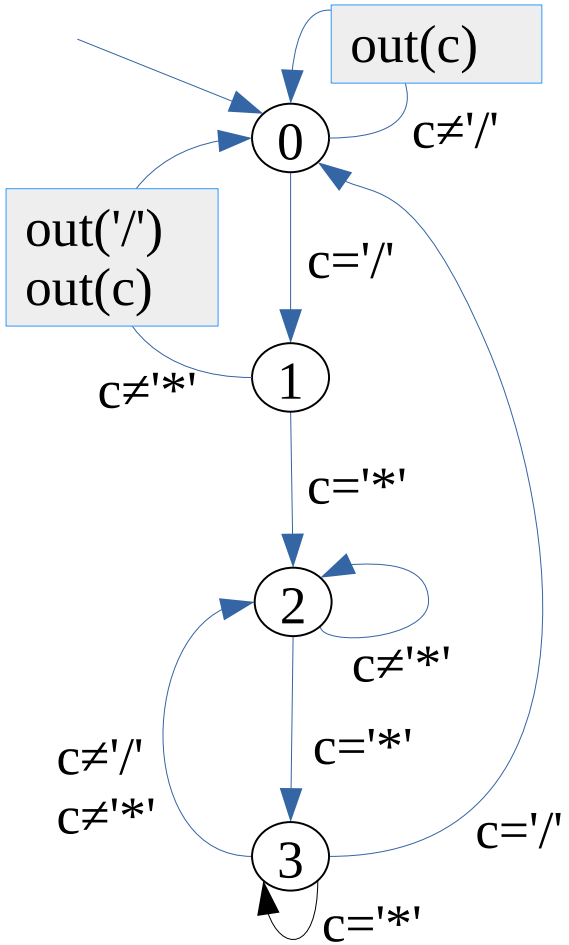
1. Le complémentaire d'un langage rationnel est rationnel
2. L'intersection de 2 langages rationnels est rationnel

Preuve ?

Petit commentaire

Supprimer les commentaires /* … */

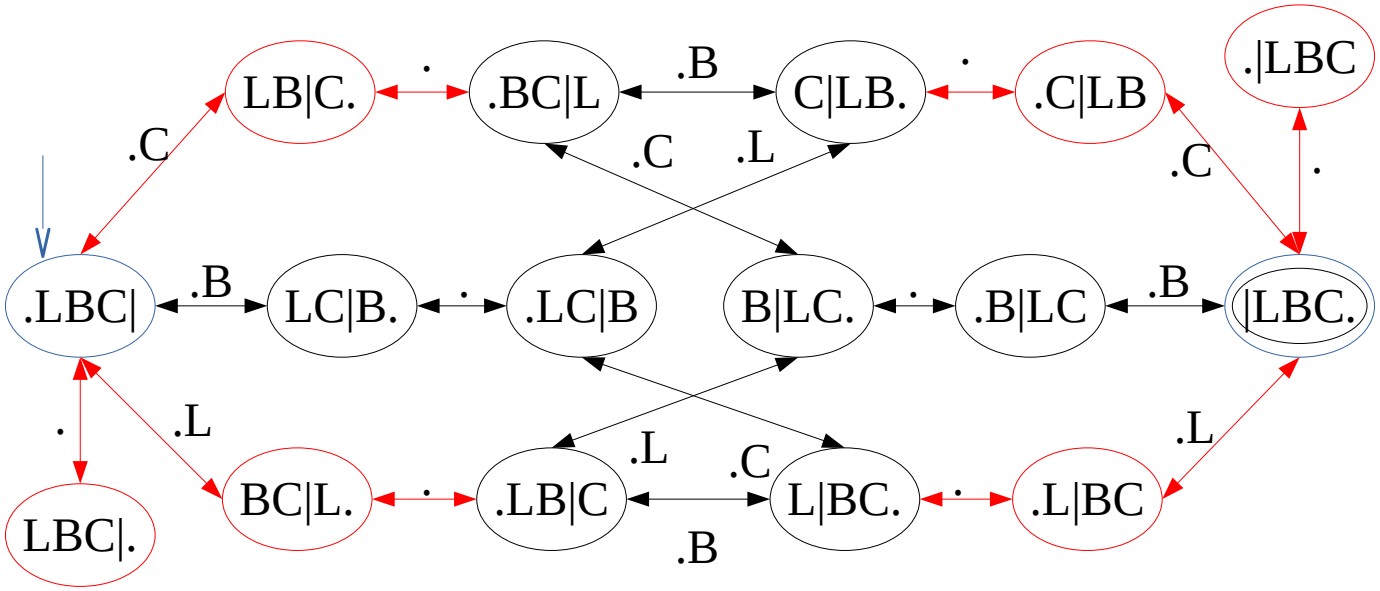
- Expression régulière
  - "/*" .* "*/" !!!
  - "/*" tout_sauf("*/") "*/" ???
- Plus facile : Automate
- Outils lex, flex, JFlex offrent aussi une vision automate : *Start-Condition*, Super-état



Illustrations (1/5)

LuBuChu

une rivière, une barque, un fermier,
un loup, un bouc, un chou

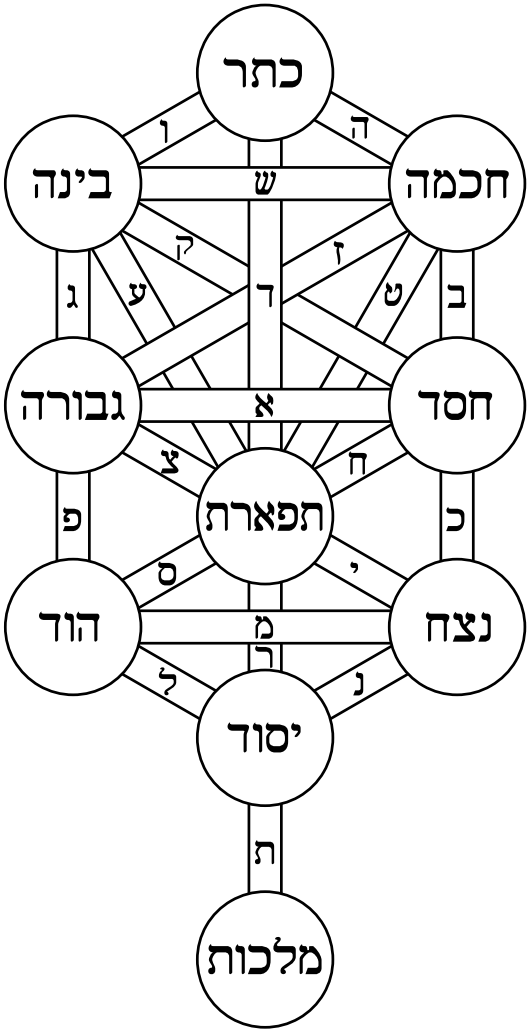


Illustrations (2/5)

L’ arbre de vie.

10 Sephiroth et 22 sentiers

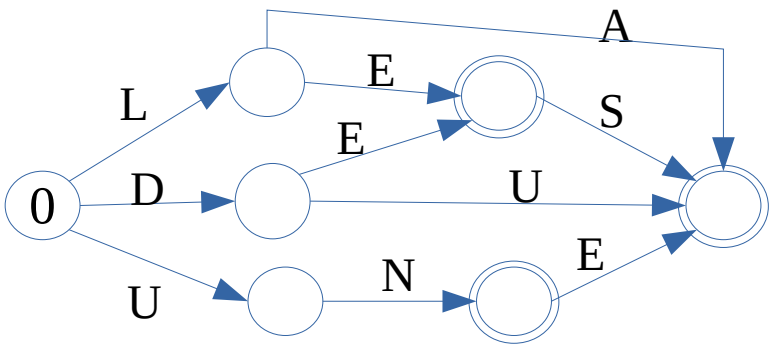
Azriel de gérone XIIIe siècle



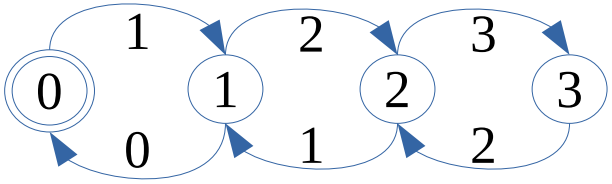
Illustrations (3/5)

Langage fini

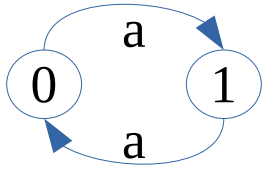
Structure de données pour dictionnaire

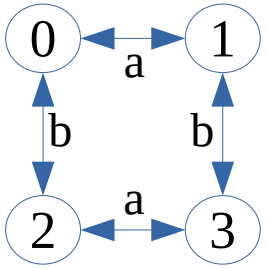


Ascenseur



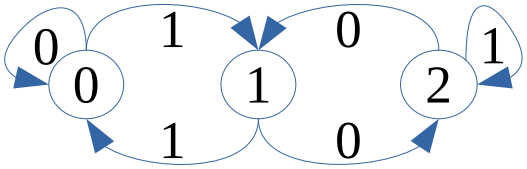
Parité



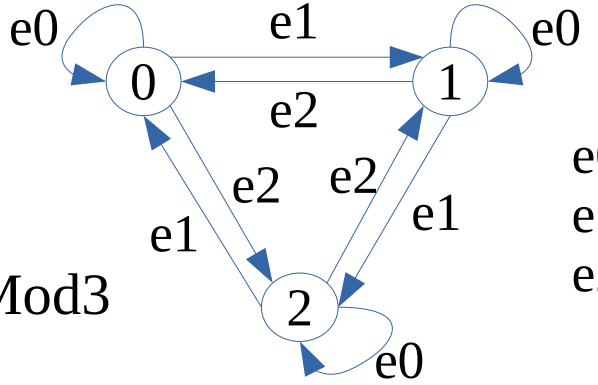


Illustrations (4/5)

Numération et Modulo



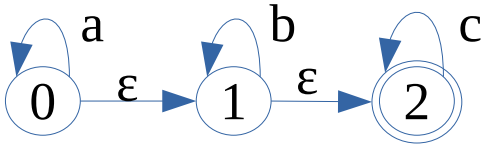
Bin-Mod3



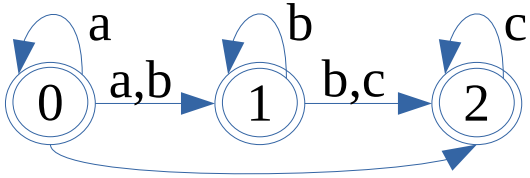
Dec-Mod3

e0= 0|3|6|9
e1= 1|4|7
e2= 2|5|8

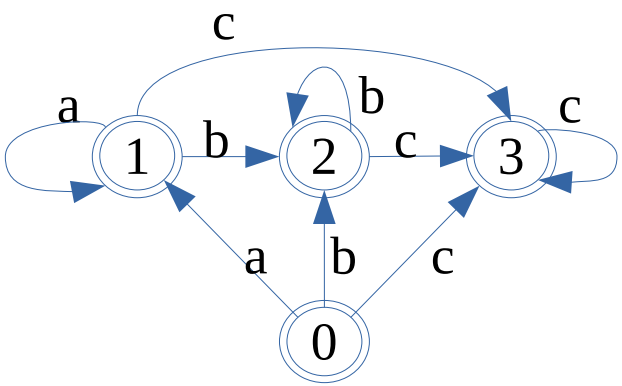
Déterminisme



Non déterministe avec ε



Non déterministe sans ε



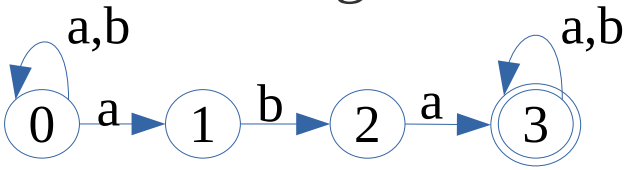
Déterministe

Illustrations (5/5)

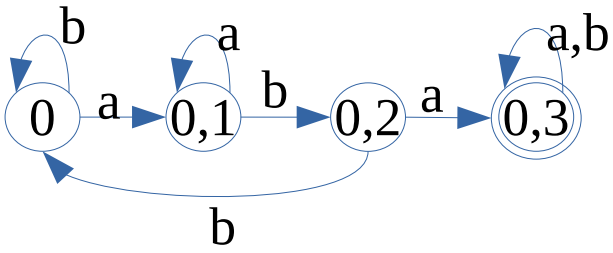
Négation de « .*aba.* »

Exercice TP ABBA-3 : preuve du corrigé

Non déterministe



Déterministe complet



Négation :

états finaux = 0, {0,1}, {0,2}

Re (0->0) = b(aa*bbb*)* = b*(a+bb+)**

*Re (0->{0,1}) = Re (0->0)aa**

*Re (0->{0,2}) = Re (0->0)aa*b*

Re (0->final) = b(a+bb+)* (ε | aa* | aa*b)*

=> Not (.*aba.*) == b*(a+bb+)*(a*|a+b)

aussi = b(a+bb+)*a*b?*

Chapitre 3

3) Grammaire Algébrique

Grammaire (1/3)

(rappel) Exemple de langage naturel élémentaire

- Lexique

POINT = .
ARTICLE = un | le
VERBE = mange | tue
NOM_C = lapin | chasseur | gnou
NOM_P = Bart | Jeannot | Obi-Wan

- Syntaxe

<Phrase> := *<SujetVerbeCompl>* **POINT** ;
<SujetVerbeCompl> := *<GroupeNom>* **VERBE** *<GroupeNom>* ;
<GoupeNom> := **NOM_P** | **ARTICLE NOM_C** ;

- Des phrases valides :
  - Obi-Wan mange un gnou.
  - le lapin tue un chasseur.

Grammaire (2/3)

Grammaire algébrique (*context free)*

$G = (V_T , V_N , S , P)$ avec

V_T, ensemble des symboles terminaux ou constantes ou t*okens*

V_N, ensemble des symboles non-terminaux ou variables

V_T et V_N sont finis et disjoints

$S \in V_N$, symbole de départ ou axiome.

$P \subset V_N \times (V_T \cup V_N)^*$, ensemble des règles de production :

$$v := s_1\ s_2\ \ldots s_n\ ;$$

- Le membre gauche **v** est un symbole non-terminal, le membre droit est une concaténation de symboles $\mathbf{s_i}$ terminaux ou non-terminaux
- Les productions peuvent être regroupées en utilisant l'alternative :

$v := s_1\ s_2\ \ldots s_n\ ;$ $v := s'_1\ s'_2\ \ldots s'_p\ ;$	$v := s_1\ s_2\ \ldots s_n$ $\quad \mid s'_1\ s'_2\ \ldots s'_p\ ;$

Grammaire (3/3)

Utilisations

- En **production**
  - en partant de l'axiome, substitutions d'un membre gauche d'une production par un membre droit.
  - on génère l'ensemble des phrases valides pour la grammaire
  - Ex :

    <Phrase> => *<SujetVerbeComp>* **POINT**

    => *<GroupeNom>* **VERBE** *<GroupeNom>* **POINT**

    => **NOM_P VERBE ARTICLE NOM_C POINT**

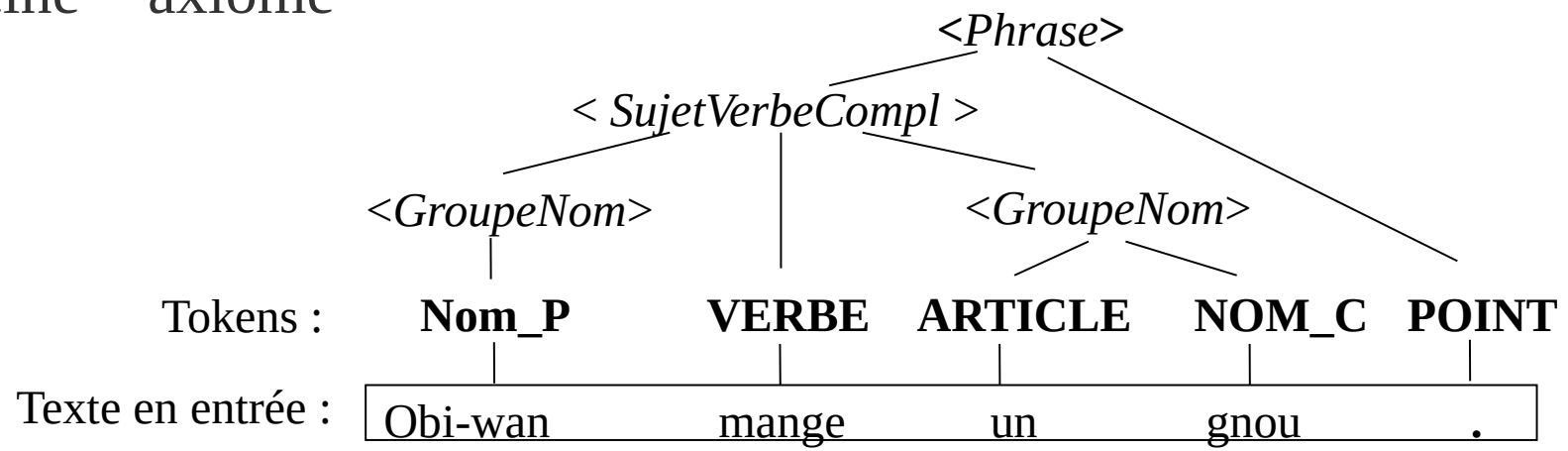
    => Bart tue un lapin .
- En **reconnaissance, *parsing***
  - Réductions d'une phrase en remplaçant des membres droits par les membre gauches
  - La phrase est valide si les réductions terminent sur l'axiome

Arbre Syntaxique

Décrit la reconnaissance d'une phrase donnée conformément à une grammaire

Arbre étiqueté :

- Feuilles = *tokens* = mots de V_T
- Nœud = membre gauche de production = symbole de V_N
- Fils = membres droits de la production
- Racine = axiome



Où est l’étoile ?

Problème !

- Les grammaires algébriques contiennent la concaténation et l’alternative mais pas la fermeture de Kleene
- Selon Chomsky, les langages rationnels sont algébriques
- Comment réaliser l’étoile avec une grammaire algébrique ?

Exercice

- Soit L un langage décrit par le symbole L, comment définir le langage L* avec une grammaire algébrique ?
  - N.B. : si L est rationnel, L* est rationnel et doit donc être algébrique

Où est l'étoile ? (soluces)

Solution(s)

« La récursivité contient l'étoile »

Récursif Droit	Récursif Gauche
Lstar := ε | L Lstar ;	**Lstar := ε | Lstar L ;**

ou aussi ! **Lstar := ε | L | Lstar Lstar ;**

- N.B. : dans les outils, le mot vide **ε** s'écrira **/* mot vide */**

Exemple

Ébauche d'un langage informatique

- Vocabulaire terminal

```
TYPE      = int | char | float
IDENT     = [a-zA-Z] [a-zA-Z0-9]*
INT       = -? [0-9]+
FLOAT     = -? [0-9]+ ( \. [0-9] *) ?
```

- Règles de productions

```
Program    :=  Definitions  Functions  ;
Definitions :=  TypeVar
            |   Definitions  TypeVar  ;
TypeVar    :=  TYPE  VarList  ';'  ;
VarList    :=  Var
            |   VarList  ','  Var ;
Var        :=  ScalarVar | ArrayVar  ;
ScalarVar  :=  IDENT
            |   IDENT '='  Number ;
ArrayVar   :=  IDENT  '['  INT  ']'
            |   ArrayVar  '['  INT  ']' ;
Number     :=  INT | FLOAT  ;
```

- Exemple de production

```
int i, j, i42=-1 ;
int a[-42], b[3][1][4] ;
float pi=3 ;
int yy=0.3 ;
```

Grammaire Régulière (1/2)

Une grammaire algébrique peut décrire un langage rationnel

- Indécidable dans le cas général (non-déterministe)
- Mais certaines productions simples génèrent des langages rationnels

Grammaire Linéaire Gauche (resp. Droite)

- Toute production contient au plus un symbole non-terminal en membre droit
- Ce symbole non-terminal est le premier (resp. dernier) symbole en membre droit

Linéaire Gauche	Linéaire Droit
$v_i := v_j\ t_1\ t_2 \ldots t_n\ ;$	$v_i := t_1\ t_2 \ldots t_n\ v_j\ ;$

Les grammaires linéaires gauches (resp. droite) engendrent les langages rationnels

Grammaire Régulière (2/2)

Exemples

1) $V_N=\{S,A\}$, $V_T=\{\mathbf{a},\mathbf{b}\}$

Expressions régulières pour *A* et *S* ?

S := **b** *A* | **a** *S* ;
A := **b** *A* | **a** ;

2) $V_N=\{S\}$, $V_T=\{$**LETTRE**, **CHIFFRE**$\}$

LETTRE=[a-z][A-Z], **CHIFFRE**=[0-9]

Expression régulière pour *S* ?

S := **LETTRE**
| *S* **LETTRE**
| *S* **CHIFFRE** ;

3) $V_N=\{S\}$, $V_T=\{\mathbf{a},\mathbf{b}\}$

Linéaire ? Régulier ?

Expression régulière pour *S* ?

S := **a b**
| **a** *S* **b** ;

Grammaire Régulière (2/2 soluce)

Solutions

1) A = b* a

S = a* b A = a* b+ a

L(S) = {ba, aba, aaba, abba, aaaba ...}

S := **b** A | **a** S ;
A := **b** A | **a** ;

2) S = [a-zA-Z] [a-zA-Z0-9]*

S := **LETTRE**
| S **LETTRE**
| S **CHIFFRE** ;

3) Linéaire « milieu », ni droite ni gauche

Non régulier,

L(S) = {$a^n b^n$, n>0}

Linéaire « et droite et gauche » :

S := **a b** ;
S := **a** T ;
T := S **b** ;

S := **a b**
| **a** S **b** ;

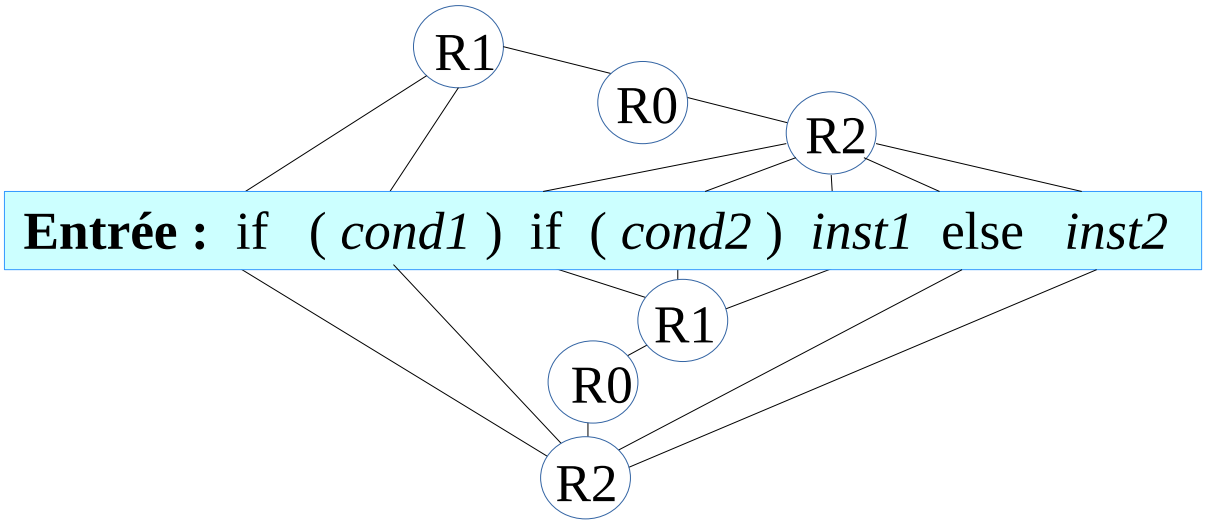
Grammaire Ambiguë (1/2)

Plusieurs arbres de syntaxe possibles pour une entrée

- Ambiguë => indéterminisme dans l'analyse syntaxique
- En général , arbres différents => sémantiques différentes

Exemple : Instruction IF

```
(R0)  Inst     :=   Inst_if | ... ;
(R1)  Inst_if  :=   "if" "(" Condition ")" Inst
(R2)           |    "if" "(" Condition ")" Inst "else" Inst ;
```



Grammaire Ambiguë (2/2)

Grammaire d'opérateur (pas de mot vide, pas de non-terminaux voisins)

Généralement ambiguë

2+3+4 = (2+3)+4 ou 2+(3+4) ?

a=1; a + ++a + ++a = ?

2-3-4 = (2-3)-4 ou 2-(3-4) ?

2+3*4 = (2+3)*4 ou 2+(3*4) ?

```
Expr    :=  Expr '+' Expr
        |   Expr '-' Expr
        |   Expr '*' Exp
        |   '-' Expr
        |   '(' Expr ')'
        |   INT
        |   SYMBOL ;
```

Solutions

- Réécriture de la grammaire
  en ajoutant des non-terminaux

- Gestion de priorités sur les règles
  dans l'analyse syntaxique

```
/* Version non ambiguë gauche */
Expr    :=  Term
        |   Expr '+' Term ;
        |   Expr '-' Term;
Term    :=  Facteur
        |   Term '*' Facteur ;
Facteur :=  '-' Expr
        |   '(' Expr ')'
        |   INT
        |   SYMBOL ;
```

Spécification avec BNF (1/2)

Syntaxes Multiples : BNF, EBNF, ABNF

Regroupe grammaire algébrique et expression régulière

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

Éléments de syntaxe

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

Spécification 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 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.)
```

Chapitre 4

4) Analyse Syntaxique

Analyse syntaxique : trois stratégies

Résolution générale

- Applicable aux langages algébriques non déterministes
- Complexité $O(n^3)$; Algorithmes : Earley, CYK,Valiant, …

Analyse descendante ou « LL »

- Applicable à « beaucoup » de langages algébriques déterministes
- Algorithme plus intuitif, mais 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) ; Outils : yacc, bison, CUP, GOLD, ...

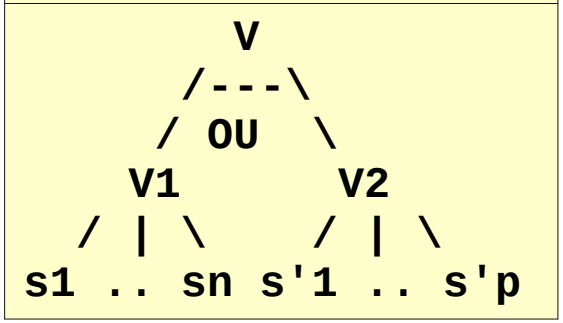
Analyse descendante (1/3)

Principe

- Construction de l'arbre de syntaxe de haut en bas
- Utilisation de la grammaire en production
- Arbre de décision ET/OU
  - Nœud OU : pour remplacer un symbole non-terminal, on a le choix entre les différentes productions (ou alternatives) avec ce symbole en membre gauche
  - Nœud ET : pour une production, il faut reconnaître la séquence des symboles en membre droit

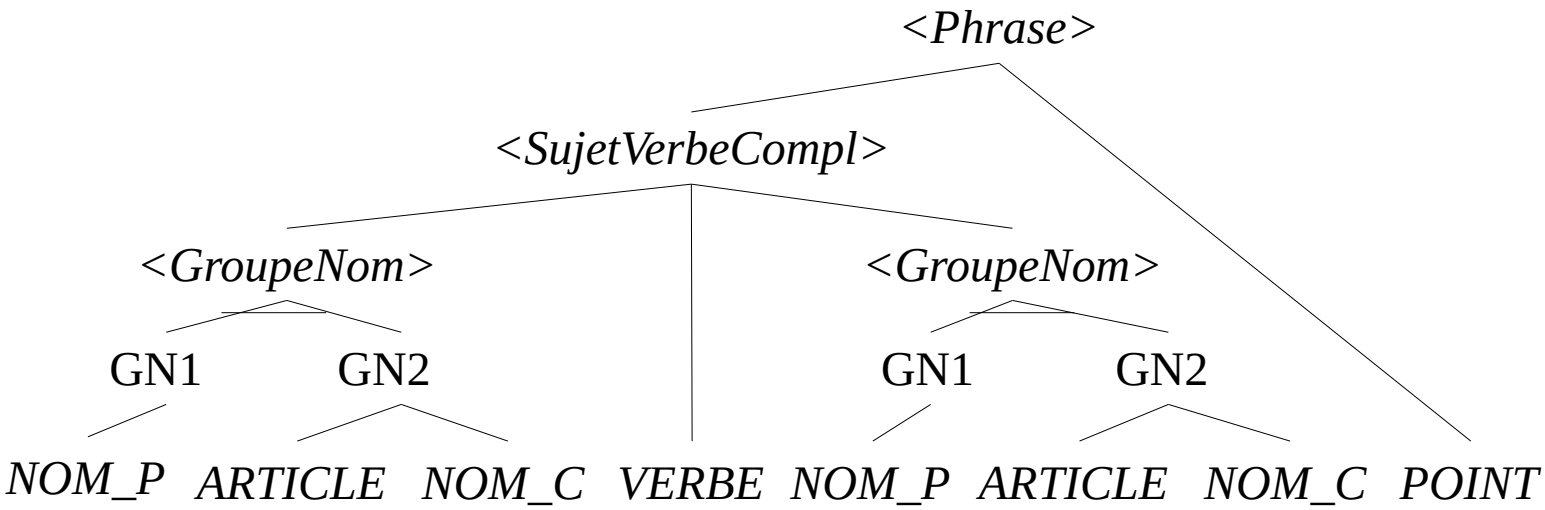
- Exploration de l'arbre
  - En profondeur avec retour arrière
  - En « parallèle » ...
  - Rendre prédictible les choix OU ?

(v1) v := s_1 s_2 …s_n ;
(v2) v := s'_1 s'_2 …s'_p ;



Analyse descendante (2/3)

Exemple langage naturel élémentaire



Parcours en profondeur : « Obi-Wan mange un gnou. » ?

- Choix GN1, Obi-wan=*NOM_P ?* ok, mange=*VERBE ?* ok,
  - Choix GN1, un=*NOM_P ?* echec, retour dernier choix,
  - Choix GN2, un=*ARTICLE ?* ok, gnou=NOM_C ? ok, .=POINT ? ok
- Fini, Accepte.

Analyse descendante (3/3)

Descente récursive : Mise en œuvre simple et intuitive

Pour chaque symbole X, une fonction reco_X()

- Si X est terminal reco_X() lit le *token* X en entrée, ou échec
- Si X est non-terminal, reco_X() choisit une production de X et appelle successivement reco_S_i() pour les symboles en membre droit

Contraintes sur l'écriture de la grammaire

Pas de récursivité gauche dans les productions !!!

directement L := L a | a ; ou indirectement L := M … ; M :=L... ;

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

Rendre prédictible (déterministe)

Utilisation du *Lookahead*, calcul de fonctions *first(),next()*

Principe : une production n'est applicable que si récursivement elle peut générer le prochain *token* en entrée

Analyse ascendante « Shift/Reduce » (1/4)

Principe

- Construire l'arbre syntaxique de bas en haut
- Utilisation de la grammaire en reconnaissance
  - Réduction = remplacement membre droit par membre gauche
- Utilisation d'une pile
  - *Shift* = empiler le *token* suivant
  - Tester les réductions possibles en tête de pile
  - *Reduce* = dépiler *n* symboles d'un membre droit, empiler le symbole non-terminal du membre gauche

Indéterminisme

- « *reduce/reduce* » plusieurs réductions possibles au même moment
- « *shift/reduce* » réduction possible, mais décalage préférable

NB : Si il existe une production vide, il y a toujours une réduction possible même avec une pile vide !

Analyse ascendante « Shift/Reduce » (2/4)

Entrée = « Obi-Wan mange un gnou. »

Opération	**Pile**
décalage	*NOM_P*
réduction	*<GroupNom>*
décalage	*<GroupNom> VERBE*
décalage	*<GroupNom> VERBE ARTICLE*
décalage	*<GroupNom> VERBE ARTICLE NOM_C*
réduction	*<GroupNom> VERBE <GroupNom>*
réduction	*<SujetVerbeCompl>*
décalage	*<SujetVerbeCompl> POINT*
réduction	*<Phrase>*
Succès	

N.B. Succès = règle conventionnelle « $START = <Phrase> EOF »

Analyse ascendante « Shift/Reduce » (3/4)

Entrée = « alpha + bêta + 666 »

Expr := *Expr* *'+'* *Expr*
| **INT**
| **SYMBOL** ;

Opération	**Pile**
décalage	**SYMBOL**
réduction	*Expr*
décalage	*Expr* **'+'**
décalage	*Expr* **'+'** ***SYMBOL***
réduction	*Expr* *'+'* *Expr*

Ambiguïté
Conflit *shift/reduce*

réduction	*Expr*
décalage	*Expr* **'+'**
décalage	*Expr* **'+'** **INT**
réduction	*Expr* **'+'** *Expr*
réduction	*Expr*
succès	

(alpha+bêta)+666

décalage	*Expr* **'+'** *Expr* **'+'**
décalage	*Expr* **'+'** *Expr* **'+'** **INT**
réduction	*Expr* **'+'** *Expr* **'+'** *Expr*
réduction	*Expr* **'+'** *Expr*
réduction	*Expr*
succès	

alpha+(bêta+666)

Analyse ascendante « Shift/Reduce » (4/4)

Input = « 42 666 »

(R0)	*Liste*	:=	/* mot vide */
(R1)		|	*Liste* **INT** ;

(R0)	*Liste*	:=	/* mot vide */
(R1)		|	**INT** *Liste* ;

Opération	**Pile**

Opération	**Pile**

- Quand fait-on la réduction R0 ?
- Dans quel ordre sont réduits les entiers de la liste ?

Analyse ascendante « Shift/Reduce » (4/4 soluce)

Input = « 42 666 »

(R0) *Liste* := /* mot vide */
(R1) | *Liste* **INT** ;

Opération	Pile
Réduction R0	*Liste*
décalage	*Liste* **INT**
Réduction R1(42)	*Liste*
décalage	*Liste* **INT**
réduction R1(666)	*Liste*

(R0) *Liste* := /* mot vide */
(R1) | **INT** *Liste* ;

Opération	Pile
décalage	**INT**
décalage	**INT INT**
Réduction R0	**INT INT** *Liste*
Réduction R1(666)	**INT** *Liste*
Réduction R1(42)	*Liste*

Récursivités droites ou gauches valides

Récursivité gauche préférée par analyseur LR

Automate LR (1/3)

Analyse LR : Automate à pile pour décider des shift/reduce

États de l'automate

- états viables du sommet de pile (séquence de *n* symboles)
- viable = le sommet de pile pourra être réduit à terme
- sommet de pile = préfixe de membre droit de production

Pile

- A chaque *Push*, l'état de l'automate est empilé avec le symbole

Transitions

- *Shift* : transition « simple » avec *Push* d'un *token*
- *Reduce* : transition avec remplacement (*Pop** + *Push*) et retour à l'état stocké dans la pile (dernier *Pop*)
- En général, les transitions *reduce* ne sont pas représentées

Automate LR (2/3)

Exemple LALR(1) avec « cup -dump »

```
terminal TOK;
nonterminal list;
list ::= /* vide */
       | list TOK
;
```

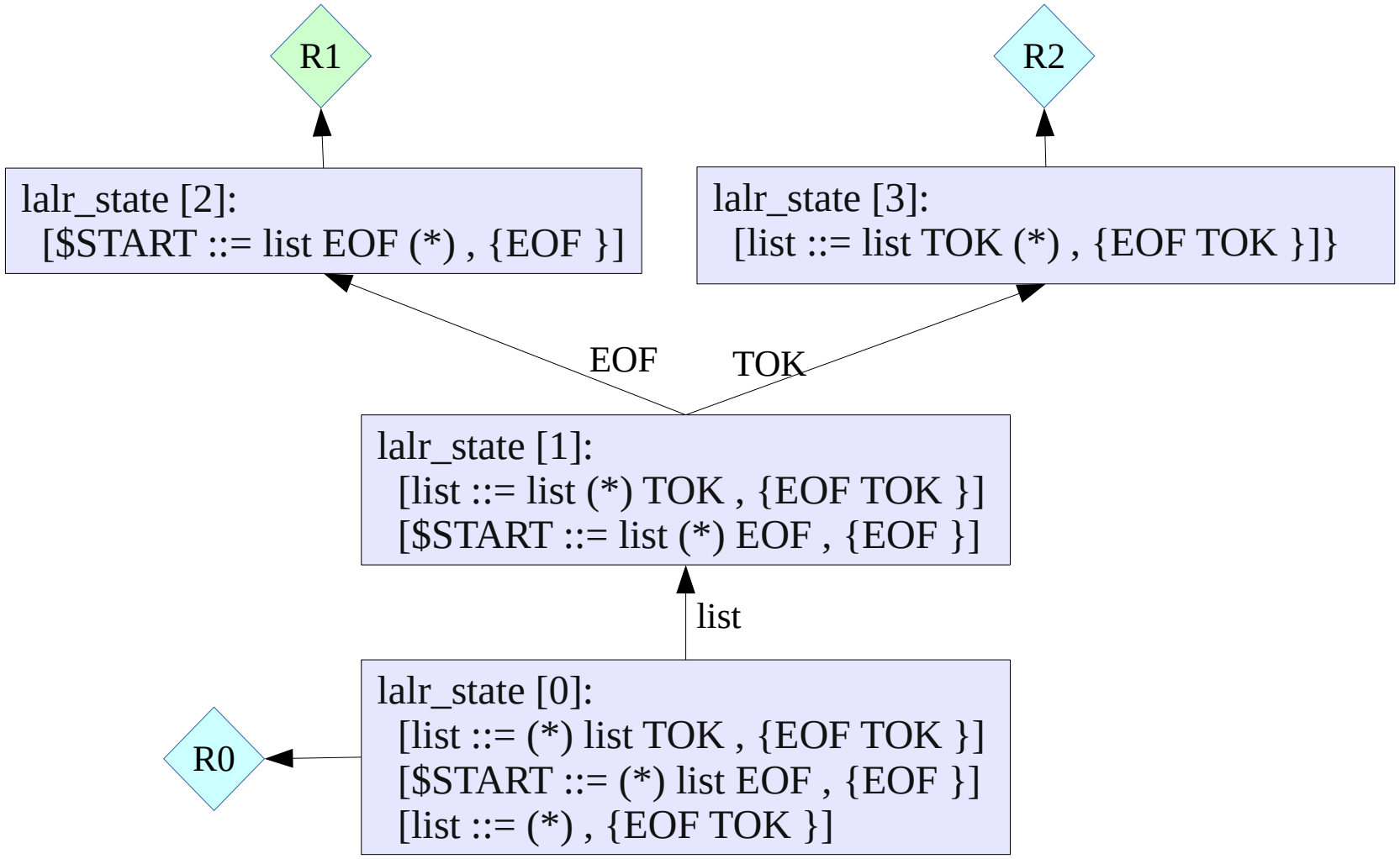
```
== Terminals ==
[0]EOF
[1]error
[2]TOK
=== Non terminals =
[0]list
=== Productions =
[0] list ::=
[1] $START ::= list EOF
[2] list ::= list TOK
```

```
=== Viable Prefix Recognizer ===
–--lalr_state [0]:
  [list ::= (*) list TOK , {EOF TOK }]
  [$START ::= (*) list EOF , {EOF }]
  [list ::= (*) , {EOF TOK }]
transition on list to state [1]
–--lalr_state [1]:
  [list ::= list (*) TOK , {EOF TOK }]
  [$START ::= list (*) EOF , {EOF }]
transition on TOK to state [3]
transition on EOF to state [2]
–--lalr_state [2]:
  [$START ::= list EOF (*) , {EOF }]
–--lalr_state [3]:
  [list ::= list TOK (*) , {EOF TOK }]
```

```
---- ACTION_TABLE ----
From state #0
 [term 0:REDUCE prod 0]
 [term 2:REDUCE prod 0]
From state #1
 [term 0:SHIFT to state 2]
 [term 2:SHIFT to state 3]
From state #2
 [term 0:REDUCE prod 1]
From state #3
 [term 0:REDUCE prod 2]
 [term 2:REDUCE prod 2]
–-- REDUCE_TABLE ----
From state #0
 [non term 0->state 1]
From state #1
From state #2
From state #3
```

Automate LR (3/3)

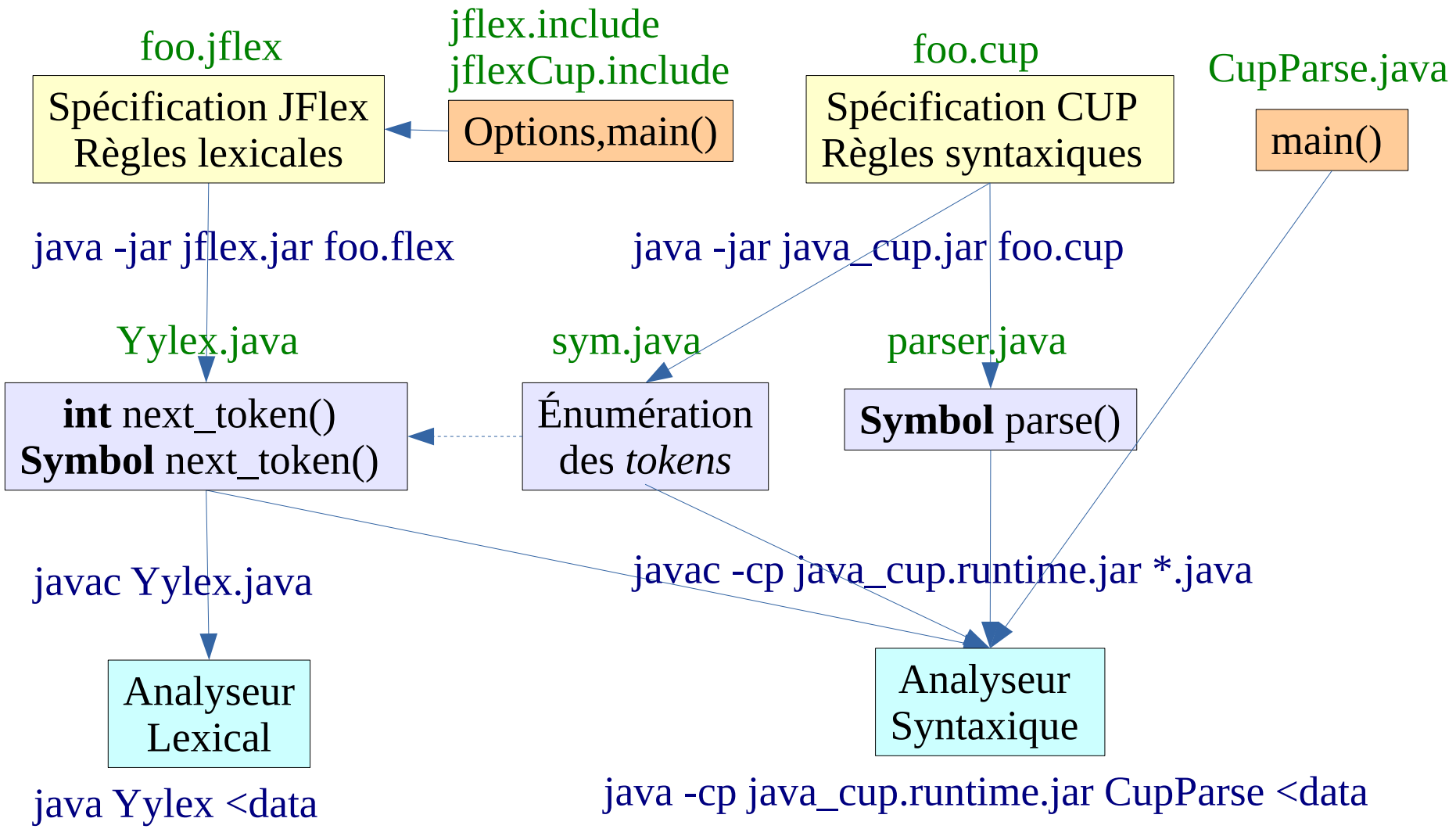
(using CUP eclipse plugin)



Chapitre 5

5) Outils JFlex et CUP (annexe)

Utilisation JFlex et CUP



Format des spécifications

Jflex

```
package cours.compilation
import cours.compilation.common
%%
%include Jflex.include
%include JflexCup.include
%{
%}
%init{
%init}
%eof{
%eof}
ANY   = [^]
BLANC = [ \t]
%%
[a-zA-Z]  [a-zA-Z0-9]* { ECHO("ID"); }
[  \t] |  {NL}    { ECHO(); }
{ANY}          { WARN("Invalid char"); }
```

Cup

```
package cours.compilation
import cours.compilation.*
parser code {:  /* … */ :}
action code {:  int i ; :};
init with {: i=0; :}
terminal int TOK ;
terminal TIK, TAK ;
nonterminal int sym, var;
precedence left TIK;
precedence right TAK ;
start with sym,;
sym ::=  sym:s TOK:n  {: RESULT=s+n; :}
       |  var:v              {: RESULT=v; :}
;
var ::=  /* vide */   {: RESULT=0;  :}
      |  TIK           {: RESULT=42 ;  i++; :}
;
```

Automate LALR et Conflit (1/2)

« cup -dump » :

```
terminal TOK;
nonterminal list;
list ::= /* vide */
      | TOK
      | list TOK
;
```

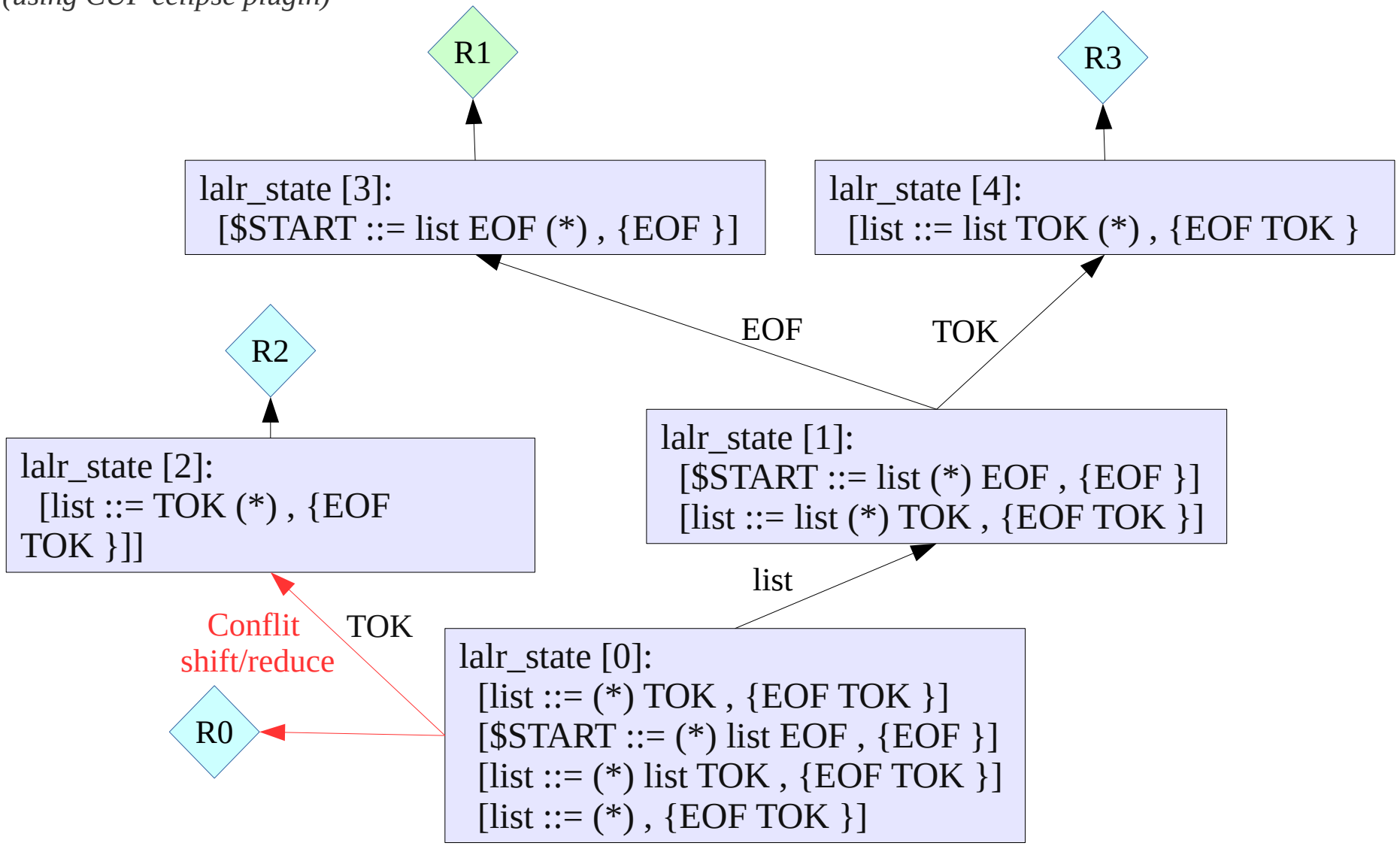
```
===== Terminals =====
[0]EOF [1]error [2]TOK
===== Non terminals ==
[0]list
===== Productions ===
[0] list ::=
[1] $START ::= list EOF
[2] list ::= TOK
[3] list ::= list TOK
```

```
Warning : *** Shift/Reduce conflict
                found in state #0
  between list ::= (*)
  and     list ::= (*) TOK
  under symbol TOK
```

```
===== Viable Prefix Recognizer ===
–--lalr_state [0]:
  [list ::= (*) TOK , {EOF TOK }]
  [$START ::= (*) list EOF , {EOF }]
  [list ::= (*) list TOK , {EOF TOK }]
  [list ::= (*) , {EOF TOK }]
transition on TOK to state [2]
transition on list to state [1]
---lalr_state [1]:
  [$START ::= list (*) EOF , {EOF }]
  [list ::= list (*) TOK , {EOF TOK }]
transition on TOK to state [4]
transition on EOF to state [3]
---lalr_state [2]:
  [list ::= TOK (*) , {EOF TOK }]
---lalr_state [3]:
  [$START ::= list EOF (*) , {EOF }]
---lalr_state [4]:
  [list ::= list TOK (*) , {EOF TOK }]
```

Automate LALR et Conflit (2/2)

(using CUP eclipse plugin)



Chapitre 6

6) Arbre de Syntaxe Abstraite

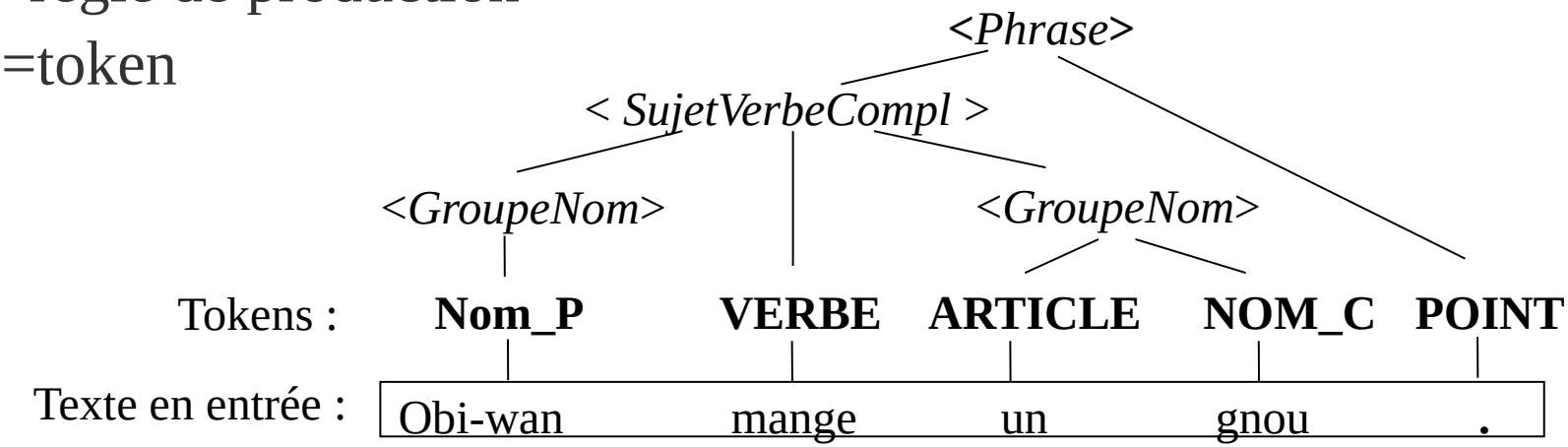
Arbre Syntaxique

L'Arbre Syntaxique est :

- La preuve de la validité syntaxique de l'entrée
- Le support d'une part importante de la sémantique de l'entrée
- La structure de données transmise dans les deux phases suivantes de la compilation : analyse sémantique et génération de code intermédiaire

Rappel :

- Nœud=règle de production
- feuille=token



...

Syntaxique

Sémantique

Bon pour génération

CST vs AST (1/2)

***Concrete Syntax Tree* (CST) :**

- Forme adaptée pour la validation syntaxique
- Pas de valeurs sémantiques pour les *tokens*
- Les nœuds respectent strictement les règles de grammaire
- Construction automatique avec les outils d'analyse syntaxique

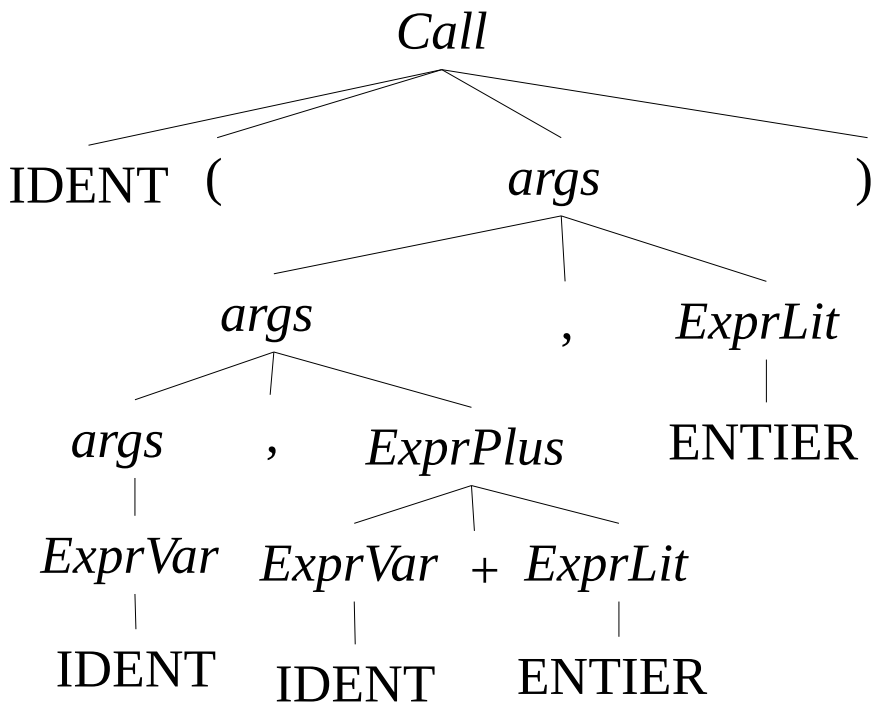
***Abstract Syntax Tree* (AST) :**

- Forme porteuse de la sémantique
- Intègre les valeurs sémantiques des *tokens*
- Supprime les *tokens* inutiles (séparateurs, parenthèses,…)
- Se libère du « tyran algébrique »
  - OPBIN vs {PLUS,MOINS,…}
  - Liste « 1 2 3 » vs List (List (List (List(ε), 1) , 2) , 3)

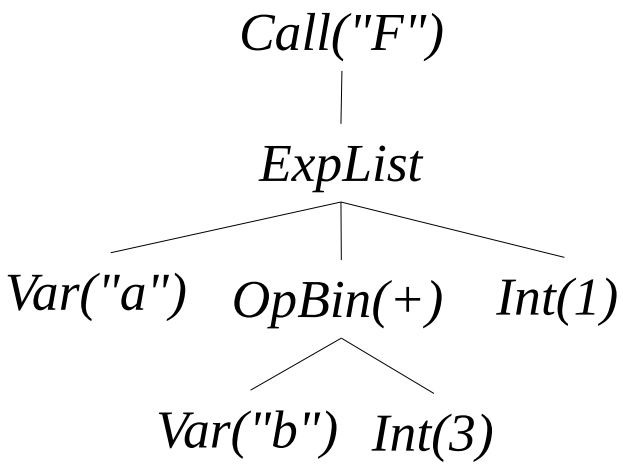
CST vs AST (2/2)

Exemples : « F(a, b+3, 1) »

CST



AST



Arbre et Analyse LR

Analyse LR

- Construction ascendante de l'arbre
- Action sémantique = nouvel arbre à partir des fils déjà existants

```
/* Exemple dans une spécification CUP */
nonterminal Arbre v,  s1, s2 , … ;
v ::= s1: x1  s2: x2  …     /* RegleN */
    {:  RESULT= new Arbre (« RegleN », x1, x2,  …) ; :}
;
/* Cas des feuilles */
temninal [type]   Tj , … ;
v ::= …Tj ...  /* RegleM */
    {:  RESULT= new Arbre (« RegleM », .. ,  new Arbre(« Tj »), ..) ; :}
;
```

Arbre et Typage Objet

Utilisation d'un langage objet :

- Typage des nœuds
  - new ArbreRegleN (x1,…) vs new Arbre(« RegleN »,x1,...).
- Héritage et productions alternatives dans la grammaire
  - ArbreV = classe abstraite pour ArbreRegleV1, ArbreRegleV2,...
- Surdéfinition (*overriding)* et liaison dynamique
  - ArbreV T = new ArbreRegleVi() ;

    T.xxx() ; /* méthode xxx de la classe concrète de T (ArbreRegleVi) */

Réalisation d'une fonction sur l'arbre

- Méthodes d'objets dans les nœuds et parcours récursifs
- Variante : patron de conception « Visiteur » (*Visitor Pattern*)

  But : obtenir le même résultat mais avec le code des méthodes d'objets de chaque nœud regroupé dans une seule classe.

Visiteur (1/2)

Ajouter une fonction à une hiérarchie de classes (classes visitées)

- Non pas en utilisant le schéma naturel OO des méthodes d'instance dans chaque classe (héritage, surdéfinition)
- Mais en utilisant une classe externe (classe visiteuse).

Utilité

- Regrouper l'algorithme de la fonction dans une même classe
- Ne pas modifier les classes visitées pour chaque nouvelle fonction

Problème de liaison dynamique : comment connaître dans la classe visiteuse, la classe concrète des objets visités ?

- Solution « pas très classe » : Imbrications de *if (x instanceof X1)*
- Solution « plus classe » : définir une méthode dans les classes visitées qui sert uniquement à connaître la classe concrète.

Visiteur(2/2)

« Accept et Visit sont dans un bateau »

```
class Visiteuse  extends Visitor

public void visit ( VisitéeA o) {
  /* maFonction cas A */ }

public void visit ( VisitéeB o) {
  /* maFonction cas B */ }
…

void maFonction ( Visitées  o) {
      o.accept(this) ;
}
```

```
class  VisitéeA extends Visitées
public void accept(Visitor v) {
     v.visit(this) ;
}
/* public void maFonction()  */
```

```
class VisitéeB extends Visitées
public void accept(Visitor v) {
     v.visit(this) ;
}
/* public void maFonction()  */
```

(Solution « plus classe » !?)

Classes du cours (1/4)

AstNode : Classe abstraite ancêtre des « visitées »

```
 /** Patron Visiteur */
  public abstract void accept(AstVisitor v);

/** Constructeur varargs */
  protected AstNode( AstNode ... fils )

/** Itérable avec for(AstNode f : node) {} */
  public Iterator<AstNode> iterator()
  public int size()

/** Impression d'un nœud */
  public String toString()

/** Impression Arbre */
  public String toPrint()
```

Classes du cours (2/4)

AstList<R> : fils homogènes et construction itérative

```
class AstList<R extends AstNode> extends AstNode {
   public void accept(AstVisitor v) { v.visit(this); }
/** Construction itérative avec ajout en fin de liste */
  public void add(R node)
}
```

Ast : Classe concrète de base (pour tests ou CST)

```
class Ast extends AstNode {
  public void accept(AstVisitor v) { v.visit(this); }

  public final String label;
  public Ast(String label, AstNode... f) { super(f); this.label = label; }
  public String toString() { return label; }
}
```

Classes du cours (3/4)

AstVisitor : Interface des classes « Visiteuses »

- Méthodes abstraites visit() pour chaque classe visitable de l'AST

```
interface AstVisitor {

  <T extends AstNode> void visit(AstList<T> n);

  void visit(Ast n);

 /* … idem pour chaque classe visitable. */

}
```

Classes du cours (4/4)

AstVisitorDefault : Visiteur générique de l'AST

- Classe mère pour des Visiteurs qui font des calculs par parcours en profondeur de l'arbre.
- Les méthodes visit() sont alors à surdéfinir (*override*). Ou pas !

```
class AstVisitorDefault implements AstVisitor {
  public void defaultVisit(AstNode n) {
    for (AstNode f : n) f.accept(this);
  }

  public <T extends AstNode> void visit(AstList<T> n) { defaultVisit(n); }
  public void visit(Ast n)    { defaultVisit(n); }
 /* … idem pour chaque classe visitable. */
}
```

(Bonus) Retour au source

Propager les informations de positions dans le fichier source

- JFlex vers CUP : transparent avec *JflexCup.include*
  - ComplexSymbolFactory et TOKEN()
- CUP vers AST :
  - Option « -locations » de CUP requise (Makefile ou build.xml)
  - Méthode addPosition() de AstNode :

```
v ::= s1: aa   …  sn:zz
    {:    RESULT= new Arbre ( aa,  …,zz) ;
          RESULT.addPosition(aaxleft, zzxright) ;   :}
;
```

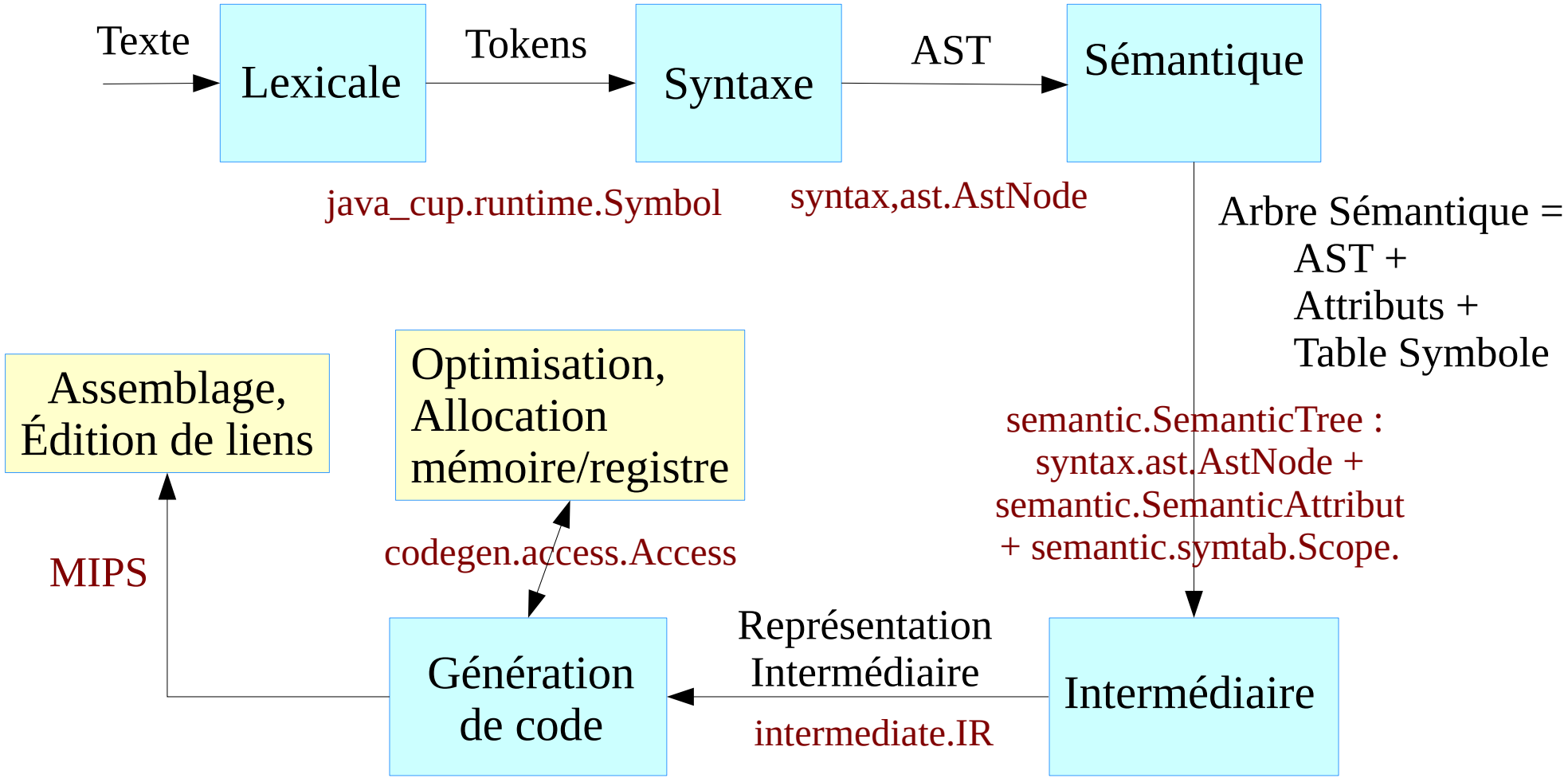
  - Exemple : AstNode.toString() -> RopBin[16/10/293-16/21/304](+)

Chapitre 7

7) Analyse Sémantique

Où en est-on ? (1/3)

Retour : Phases de Compilation



Où en est-on ? (2/3)

Compilation Classique versus Moderne

- « Dragon Book » Aho, Ullman 1977
- « Modern Compiler » Appel 2000 , ...

Identique pour l’analyse lexicale et syntaxique

Différences importantes sur l’analyse sémantique et la génération de la forme intermédiaire

Différences sur les solutions techniques et aussi sur le problème à résoudre

Où en est-on ? (3/3)

Compilation Multi-passe ou Mono-passe ?

- Exécution en séquence vs chaînage (pipeline) des phases
- Mono-passe : contraignant sur l'algorithmique du compilateur, mais économe sur les performances en temps, en espace, en entrée-sortie
- Multi-passe : plus souple dans la programmation et l'indépendance des phases de compilation

Historiquement : Mono-passe était un enjeu vital

- Contrainte sur le langage : Tout ce qui est nécessaire pour analyser une portion de programme doit exister « avant » dans le source.
- Algorithmique lourde pour tout résoudre en une fois

Aujourd'hui : Multi-passe encouragé

- Contraintes plus faibles sur les performances
- Langages plus «libéraux» (i.e. Java)

Analyse sémantique (1/2)

Objectifs :

- Valider les règles du langage non gérables au niveau syntaxique
- Établir la signification du programme et sa sémantique d'exécution
- Construire les éléments de contexte : Informations nécessaires pour l'exécution mais non explicites dans l'arbre de syntaxe, ou explicites dans une autre partie de l'arbre
- Vérifier ou Inférer des propriétés de l'algorithme : Invariants
  - Terminaison, validité de l'algorithme, non débordement, …
  - N.B. : Théorème de Rice => problèmes indécidables

Analyse sémantique (2/2)

De la théorie :

- « Haute » : Lambda-calcul, Théorèmes du point fixe, Preuve de programmes
- « Basse » : Grammaires Attribuées, Traduction dirigée par la syntaxe

Mais en pratique ?

- Trop théorique, et peu de résultats pour de « vrais » programmes
- Trop orientée sur la contrainte « compilation mono-passe »

Conclusion : Approche pragmatique

- Des fonctions sémantiques réalisées de façons modulaires et en passes successives

Attributs sémantiques

Analyse Sémantique = Décoration de l'AST

- Calcul d'attributs associés aux différents nœuds de l'arbre de syntaxe
  - Visibilité des identificateurs
  - Type de données
  - Chemins d'exécution
  - …

Attribut Synthétisé vs Attribut Hérité

- Hérité : la valeur est construite à partir de la valeur du père.
- Synthétisé : la valeur est construite à partir des valeurs des fils.
- Mixte ? Essayer de décomposer en Hérité + Synthétisé
- N.B. Analyse syntaxique LR => synthétisé facile, hérité difficile

Algorithmie : Parcours récursif en profondeur d'Arbre

- cf. Mémento du projet Minijava pour la réalisation.

Analyse statique ou dynamique

Statique

- Les attributs sémantiques sont calculés sur la structure statique de l'AST.

Dynamique

- On ajoute à l'AST, les chemins d'exécutions du programme pour calculer des attributs qui vont hériter de nœuds traversés « avant » ou « après » à l'exécution.
- Algorithmique :
  - « Couture d'arbre »
  - Interprétation symbolique
  - Équations de flots de données
  - Analyse de vivacité

Des fonctions sémantiques (1/3)

Liaison des identificateurs

- Table des symboles
- Détections : « Non défini », « déjà défini », « initialisation »
- Consistance des définitions : détection de boucle (héritage Java, définitions régulières JFlex, …), …
- Paradigmes Objets : héritage, redéfinition (*override*), polymorphisme, liaison dynamique, héritage multiple .

Contrôle de Type

- Typage des expressions, contraintes d'affectation et d'appel
- Surcharge des opérateurs
- Transtypage (*cast*) implicite, équivalence des types
- Héritage Objet

Des fonctions sémantiques (2/3)

Propagation de constantes

- Évaluation «*immediate*» de l'assembleur, optimisation d'expressions
- Calcul d'intervalles : réduction des contrôles à l'exécution, optimisation de boucle, ..
- Élimination de code

Analyse de vie, Analyse de dernière définition

- Optimisation mémoire : registre / sauvegarde de registre / pile /tas
- Variable *unused*, i.e. paramètre d'appel
- *Garbage Collect*

Optimisation appels de fonctions

- Fonction vs recopie/insertion de code
- Détection / Optimisation de récursivité

Des fonctions sémantiques (3/3)

Vivacité, Analyse de flot de données/contrôle

- Code mort
- Existence « *return* » dans toutes les exécutions d'une méthode
- …

Signalement de sémantiques douteuses : Compromis entre optimisation silencieuse et détection d'erreurs de programmation

- Boucles infinies, code mort, ...
- Transtypages « étranges »
- Expressions à valeurs triviales

Table des Symboles

Liaison entre déclarations et utilisation des identificateurs

- Et mise en œuvre de la surcharge, polymorphisme, …

Attribut « mixte »

- Une déclaration est synthétisée pour trouver sa portée (*scope*)
- La visibilité est héritée, et éventuellement surchargée dans l'AST

=> Structure spécifique : « Table des symboles »

- Regroupe l'ensemble des déclarations d'identificateurs, leurs portées et leurs visibilités depuis chaque nœud de l'arbre
- Propagée dans la suite de la compilation (y compris l'exécutable)

Dans le cas d'un langage OO

- la visibilité est aussi hérité en suivant l'arbre d'héritage des classes qui est indépendant de l'AST. (information de contexte)
- Et aussi liaison dynamique, ..

Contrôle de type (1/2)

Langage compilé <-> Langage fortement typé

Typage

- Partie importante en volume de la définition d'un langage
- Mais beaucoup de règles rapides à vérifier, bien que difficilement gérables au niveau syntaxique.

Valider les contraintes de typage du langage

- Conformité des opérateurs
- Affectations
- Appel de fonctions

Fixer les contraintes pour l'implantation des variables :

- Type => taille mémoire, type de registre, …
- Construction d'un attribut synthétisé

Contrôle de type (2/2)

Valider et réaliser les transtypages implicites ou explicites:

- Équivalence de types, héritage,
- Fonctions de conversion

Calcul de la sémantique des expressions

- Surcharge des opérateurs

Gérer les mécanismes de construction de type du langage

- Références, tableaux, enregistrements/structures, énumérations, ensembles, …
- Héritage multiple

Chapitre 8

8) Représentation Intermédiaire

Objectif

Objectif génie logiciel

- Modularité du compilateur => Interface entre :
  - Partie Avant dépendante du langage source
  - Partie Arrière dépendante de la machine cible (assembleurs ou autres)

Objectif programmation

- Étape intermédiaire pour la phase « Génération »
- Génération = traduction de l'AST vers code machine
  - Linéarisation du code
  - Génération de l'assembleur
  - Allocation mémoire
  - Optimisations : allocations des registres, évaluation des expressions, ...

Principe (1/2)

Génération de la représentation intermédiaire

- Convertir les nœuds de l'AST en utilisant un nombre réduit d' « instructions »
- Utiliser des instructions « intermédiaires » entre le langage source et le langage cible
- Linéarisation (canonisation) :
  - Traduire la sémantique d'exécution sous forme de séquence d'instructions
  - Traduire en particulier les structures algorithmiques et les appels de fonction sous forme de code séquentiel avec des sauts
  - Les expressions peuvent être complètement linéarisées (« classique ») ou conserver la structure d'arbre (« moderne »)
  - N.B. : On fait implicitement de la « couture d'AST » et donc possibilités de faire de l'analyse sémantique dynamique après la forme intermédiaire

Principe (2/2)

Représentation Intermédiaire

- Instructions
  - Affectations ou *Copy*
  - Labels pour sauts et appels de fonction
  - Sauts inconditionnels et conditionnels
  - Expressions (arbres ou linéarisées)
  - Gestion des Appels : ...
- Variables
  - Héritées de l'AST
  - Ajoutées pour la forme intermédiaire : évaluation des expressions, gestion d'appels, ...
- Informations administratives
  - Définition des symboles IR : variables, labels, constantes/*immediate*
  - ...

Code à trois adresses

Programme

- Séquence d'instructions
- Linéarisation complète des expressions

Instruction

- Une opération et zéro à trois « adresses »
- « adresses » ou variables intermédiaires
  - Variables du programme source (ou nom de fonction, ou …)
  - Variables temporaires pour évaluation des expressions
  - Labels pour des sauts
  - Valeurs constantes

[OP , X , Y, Z]

« Z = X OP Y »

Exemple :

A = 2*x +1

```
[ * ,  2,  x,  t1 ]
[ + ,  t1,  1,  t2 ]
[ = ,  t2,    ,  A  ]
```

```
t1 = 2 * x
t2 = t1+1
A = t2
```

Représentation pour Minijava (1/2)

Représentation Intermédiaire : classe intermediate.IR

- Programme = séquence de IRquadruple
  - Construction itérative par ajout de IRquadruple
- Création de variables intermédiaires :
  - IRvariable newTemp(String method), IRvariable newConst(int value),
  - IRvariable newLabel(), IRvariable newLabel(String label)

Instructions : package intermediate.ir.*

- abstract IRquadruple = [op, arg1, arg2, result]
- main.EnumOper op : opérateur, uniquement nœuds *Assign*
- IRvariable result, arg1, arg2 : variables intermédiaires
- Les variables de l'AST (semantic.symtab.infoVar) implémentent IRvariable

Représentation Minijava (2/2)

Instructions (suite) :

- Affectation : QCopy
- Labels
  - QLabel et QLabelMeth
- Sauts
  - QJump : inconditionnel
  - QJumpCond : conditionnel
- Gestion d'appel
  - QCall, QCallStatic, QParam, QReturn (et QlabelMeth)
- Expressions
  - QAssign, QAssignUnary, QNew
- Expressions avec Tableau
  - QAssignArrayFrom, QAssignArrayTo, QNewArray, QLength

Traduction AST vers IR (1/4)

Génération de la représentation intermédiaire

- Classe **intermediate.Intermediate**
- Visite récursive en profondeur de l'AST
- Utilisation d'un attribut synthétisé de type **IRvariable**
  - Requis pour les nœuds expressions de l'AST (Expr*) ,
  - Contient la variable (temporaire ou constante ou AST) utilisée dans la représentation intermédiaire pour stocker le résultat de l'expression
  - Les méthodes setVar() et getVar() affecte et retourne l'attribut IRvariable associé à chaque nœud de l'AST
- Traduction et linéarisation
  - Concerne l'ensemble des nœuds de l'AST, à l'exception de quelques nœuds uniquement déclaratifs (Klass, Formal,…)

Traduction AST vers IR (2/4)

Schémas de traduction

- Expressions
  - N.B. : toutes les « traduction(exp) » exécute un setVar(exp,...)

** ExprLiteralInt ** 666	setVar node, newConst(666))
** ExprLiteralBool ** false	setVar node, newConst(0) ** ou 1 pour true **
** ExprOpBin ** exp1 + exp2	traduction(exp1) traduction(exp2) setVar node, newTemp() QAssign +, getVar(exp1), getVar(exp2), getVar(node)

…

Traduction AST vers IR (3/4)

Schémas de traduction(suite)

- Déclaration et Appel de méthode

```
/* Method */
xxx(….) {
    contenu
    Return exp }
```

```
QLabelMeth  newLabel(«xxx»)
traduction(contenu)
traduction(exp)
QReturn  getVar(exp)
```

```
/* ExprCall */
exp.xxx(arg1,arg2,..)
```

```
traduction(exp)
traduction(arg1)
 /* …  args  suivants */
QParam getVar(exp)       // this
QParam getVar(arg1)
 /* …  args  suivants */
QCall newLabel(«xxx»), newConst(nb_args)
```

Traduction AST vers IR (4/4)

Schémas de traduction (fin)

- Instructions

```
/* StmtPrint */
System.out.println(exp)
```

```
traduction(exp)
QParam getVar(exp)
QCallStatic  newLabel(«_system_out_println»), «1»
```

```
/* StmtAssign */
X = exp
```

```
traduction(exp)
setVar(node, lookup(« X »))
QCopy getvar(exp), getvar(node)
```

```
/* StmtWhile */
while (exp) inst
```

```
QLabel  L1
traduction(exp)
QJumpCond  L2,  getVar(exp)
traduction(inst)
QJump L1
QLabel L2
```

```
L1 = newLabel()
L2 = newLabel()
```

…

Chapitre 9

9) Génération de code

Différentes fonctions (1/3)

Optimisation de la représentation intermédiaire

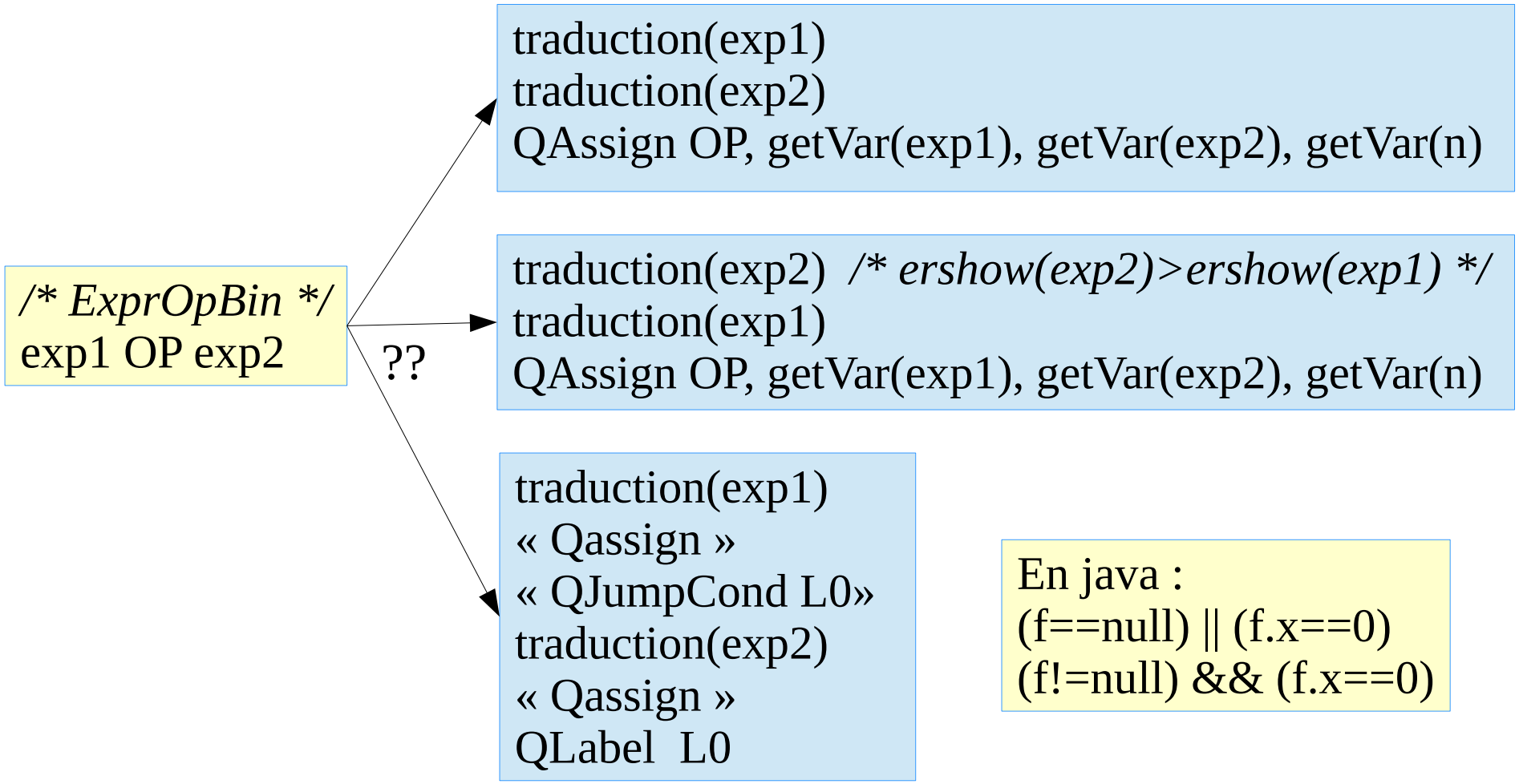
- Réduire le nombre de variables temporaires nécessaires
- Optimiser le calcul des expressions : ordre d'évaluation, recopies
- Optimiser les séquences d'instructions et les recopies
  - Graphes de flot de contrôle

Allocation Mémoire et Registre

- Définir les « adresses » pour implanter les variables (ou les méthodes avec liaison dynamique)
- Identifier la vivacité des variables :
  - Utilisation des registres ?
  - Localité et besoin de *save/restore* pour les appels de fonctions ? ..
- Optimisation des registres
  - Algorithmes de coloriage de graphe

Différentes fonctions (2/3)

Exemple d'optimisation d'expression



Différentes fonctions (3/3)

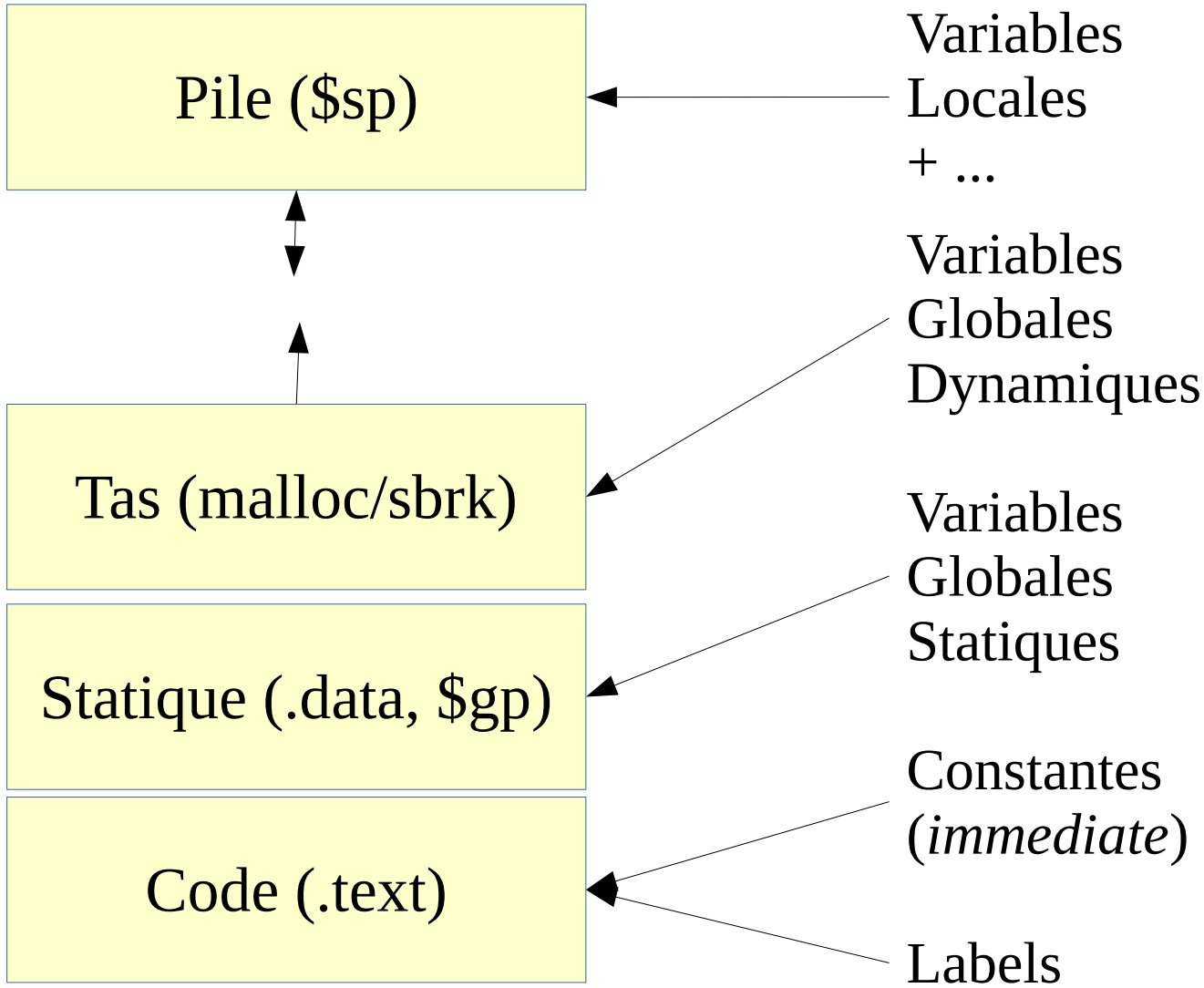
Traduction de la Représentation Intermédiaire vers le langage machine

- Réalisation d'un schéma d'appel pour les fonctions :
  - Passage d'argument, valeur de retour, adresse de retour
  - Sauvegarde de registres
  - Convention d'appel entre appelant et appelé
  - Utilisation de la pile et/ou des registres
- Sélection des instructions assembleur
- Génération, assemblage, édition de liens

Optimisation de code assembleur

- Remplacement de séquences d'instructions
  - Optimisation à lucarne

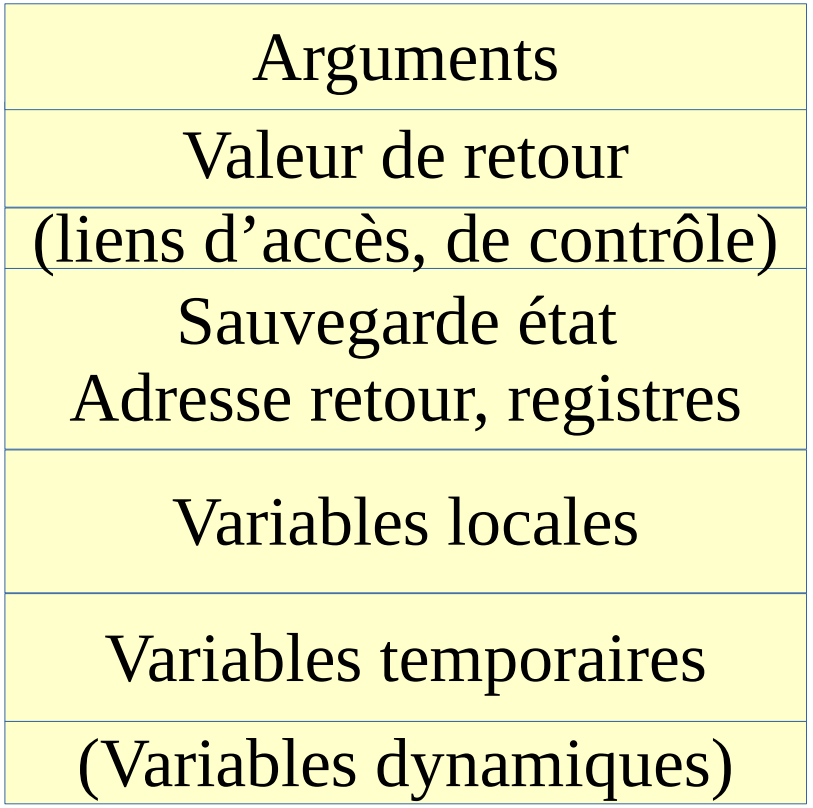
Mémoire et Variables



Cadre d’appel

Cadre d’appel = Bloc d’activation = *Frame*

- Utilisation de la pile pour les appels de fonctions
- Des choix possibles :
  - Pile ou Registre (args, …)
  - Définition de l’état (quels registres ?)
- Séquence d’appel
  - Allocation dans la pile
  - Remplissage de champs
- Séquence de retour
  - Restauration état (pile et registres)
- => **Convention d’appel**
  - Définition détaillée du cadre et Répartition entre appelant et appelé



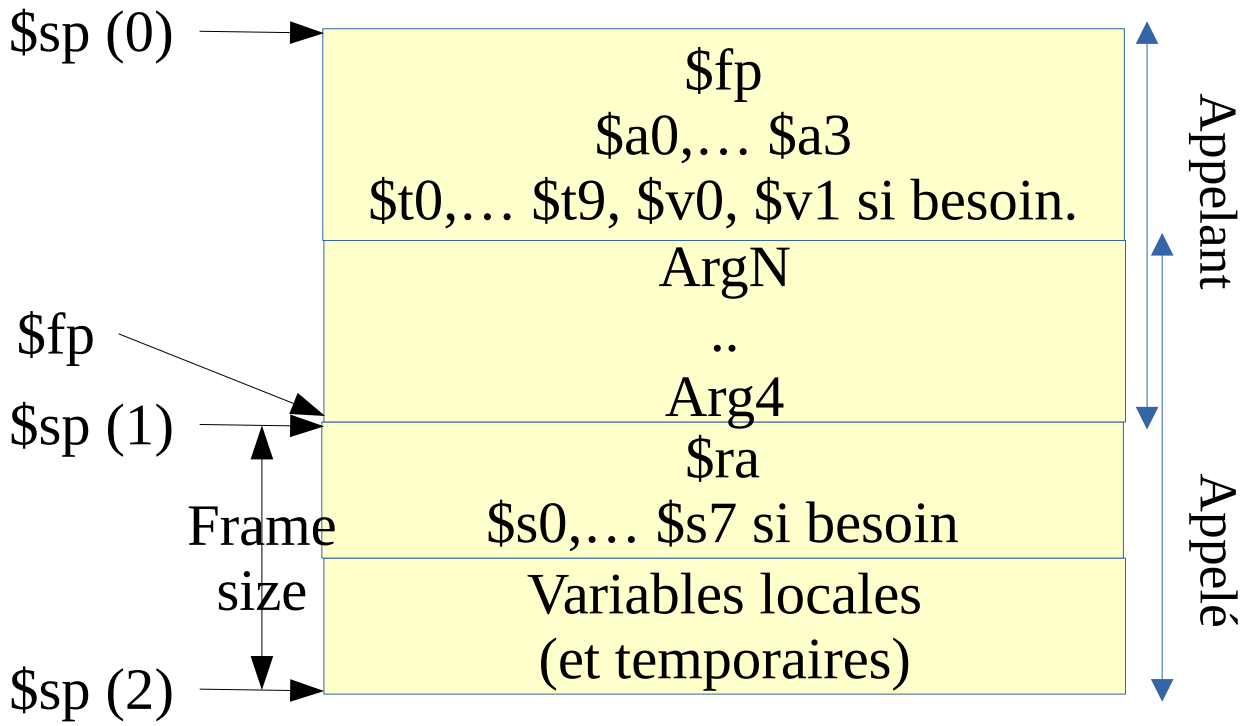
Convention d’appel MIPS/Minijava

(0) début de l’appel par l’appelant,
(1) fin de l’appel chez l’appelant
(2) début de l’appel chez l’appelé

$sp = *stack pointer*
$fp = *frame pointer*

Arguments 0 à 3
$a0,... $a3

Valeur de retour
$v0



Fin de Compilation !